Source: https://it.kgsu.ru/Lisp/oglav.html

Notes: This document is a translation of a Russian web page about LISP. It has been translated because it contains good information about the practical application of the muLISP-81, muLISP-85 and muLISP-87 systems, which had been developed by Soft Warehouse in Hawaii. The last version of muLISP was muLISP-90. muLISP was also used to create the well-known math program “Derive”.

This document has been created by automatic translation and concatenation of the individual “Step” pages of the source web site. Therefore, some key words in the code examples may have been translated so that the code has become corrupt. Typical artefacts of the translation are missing spaces between variables, extra spaces around periods, colons and commas or missing spaces when describing dotted notation. The Russian labels in the figures have not been translated.

I corrected many, but not all of these problems as I was running across them. Please keep this in mind when trying out the examples. Nevertheless, the text is very helpful for learning LISP and especially muLISP.

The copyright of the original text remains, of course, with the original authors.

Martin Hepperle, 2024

The LISP programming language

By Mikhail Vladimirovich Shvetsky and Arkady Andreevich Medvedev

Inhalt

Step 1.
Introduction to LISP language

In this step we will talk a little about the basics of the **LISP** programming language.

First, let us note that the **LISP** language is based on the lambda calculus. Let us look into the dictionary [1]: "***The lambda calculus*** is a formalism for representing functions and ways of combining them. Together with its equivalent, ***combinatorial logic***, which does not use variables, it was proposed around 1930 by logicians Church, Scheinfinkel, and Curry."

Examples of expressions:

- **lambda x.x** - the identity function with a value simply equal to its argument;
- **lambda x.c** is a constant function with value equal to c, regardless of the argument;
- **lambda x.f(f(x))** is the composition of a function f with itself, i.e. a function with a value equal to f(f(x)) for an argument x.

The wide possibilities of the considered method of designation are largely explained by the possibility of representing functions of higher orders with its help.

For example, the notation **lambda f.lambda x.(f(x))** corresponds to a (higher-order) function whose value for argument x is equal to the function obtained by composing f with itself.

Lambda calculus provides not just a notation, but also rules for equivalent transformations of lambda expressions. The most important is the **betta-** ***reduction*** rule, which can be used to simplify expressions of the form:

```
    lambda (x.e1)(e2)
```

For example, the expression (lambda x.f(x,x))(a) is betta-reduced to f(a,a).

When combined with some simple functions, the lambda calculus provides an original way of defining the set of all effectively computable functions of non-negative integers - hence, in this sense, ***it is equivalent to the Turing machine and the apparatus of recursive function theory***.

Lambda calculus is used in computer science to develop a whole class of programming languages (in particular, LISP, PAL, POP-2). Moreover, the mathematical theory developed as the first set-theoretic model of lambda calculus served as the basis for the so-called ***denotational semantics*** of programming languages."

H. Barendregt writes [2, p. 16]: "Turing's analysis shows that, despite its very simple syntax, ***the lambda calculus*** is capable of representing all mechanically computable functions. Therefore, the lambda calculus can be viewed as a paradigmatic programming language. Of course, this does not mean that we should write real programs in it. It is only implied that many problems that arise in programming, especially in connection with procedure calls, appear in the lambda calculus in a pure form. Their study can be useful in the design and analysis of programming languages.

For example, a number of programming languages have (perhaps without the direct intention of their creators) features inspired by the lambda calculus. In Algol 60, Algol 68, Pascal, procedures can be arguments of procedures. In Lisp, in addition, a procedure can be the result of a procedure."

J. Summit, a well-known specialist in programming languages, once said that all programming languages can be roughly divided into two classes. One contains **LISP**, the other contains all other programming languages [3, p.6].

The LISP language was developed at Stanford under the supervision of ***J. McCarthy*** in the early 1960s. According to the original plans, it was to include, along with all the capabilities of Fortran, tools for working with matrices, pointers, pointer structures, etc. It was assumed that the first implementations would be interpretive, but in the future, compilers would be created that would translate **LISP** constructs into machine code. Fortunately, there were not enough funds for such a project. In addition, by the time the first **LISP** interpreters were created, the dialog mode had begun to enter computer work practice, and the interpretation mode naturally fit into the general structure of dialog work. Around the same time, the principles that formed the basis of the **LISP** language were finally formed: the use of a single list representation for programs and data, the use of expressions to define functions, and the bracket syntax of the language. The process of developing the language was completed with the creation of version **LISP** 1.5, which determined the path of its development and improvement for many years [4].

Thus, **LISP** is an interpretive system, and this makes it possible to significantly simplify and speed up the process of creating complex program complexes in interactive mode, since it ensures immediate response of the system to changes made by the user, and provides powerful tools for debugging and editing programs.

“Lisp was not just a language to be used for certain purposes,” said Paul Abrahams, McCarthy's graduate student during the development of the new language, “but something to be admired as a beautiful thing. So there was a constant tension between those who admired Lisp for its purity and those who wanted to use it for various computations. Of course, a lot of computations have been done with Lisp. But in the beginning it was not so. It was often said that the main purpose of Lisp was to do more Lisp.”[3]

Based on the basic set of primitives (CAR, CDR, CONS, COND, ATOM, EQ), **LISP** is a low-level language. And from this point of view, it can be considered an assembler oriented to work with list structures. Therefore, throughout the existence of the language, there were many attempts to improve it by introducing additional basic primitives and control structures. However, all the changes, as a rule, were not grafted as independent languages. And there are several reasons for this. On the one hand, in most cases, the creators of new languages remained in the "Lisp" paradigm, without offering a new view on programming. On the other hand, new languages, as a rule, did not have their own programming environment, but "lived" in **the LISP** environment and therefore were perceived as part of this language. In its new editions, **LISP** quickly "assimilated all the valuable inventions of competitors." Finally, the authority of the Stanford School and personally J. McCarthy in the field of artificial intelligence, as well as its introduction as a mandatory course for students in all US educational institutions related to the issue of artificial intelligence, played an important role in the dissemination **of** the LISP language and its establishment as the main language of intelligent systems.

LISP, as a representative of functional programming languages, has truly amazing features.

The first most striking feature is the equivalence of the representation of programs and data in the language, which allows data structures to be interpreted as programs and programs to be modified as data. A programming method in which data external to a program is used to control the operation of the program or is itself interpreted as a program is called ***data-driven programming***. In data-driven programming, programs are stored together with data or with mappings of their types. Thus, the functions currently needed can be determined or found from the data. In **LISP,** data-driven programming is easily applicable due to the uniform form of data and program representation.

The second surprising feature is the use of recursion as the main control structure, not iteration (loop), as in imperative programming languages.

The third feature is the extensive use of the linked list data structure, so list processing is the basis of most **LISP** algorithms.

The simplicity of the LISP syntax is both an advantage and a disadvantage. A novice programmer can learn the basic rules of syntax in a few minutes, and then writing a program is mostly reduced (syntactically) to correctly writing the arguments for each function call. The problem with this simple syntax is parentheses. Each expression must have all the necessary parentheses, and since the body of any function is one giant expression, a function definition often accumulates 10-15 levels of nested parentheses. Such constructs are extremely difficult to read and debug, and an error in the placement of parentheses is probably the most common syntax error in the **LISP** language. Unfortunately, incorrect placement of parentheses in many cases is not formally considered a syntax error (i.e., it cannot be detected during compilation), the erroneous expression often looks "sensible", but its semantics (meaning) does not coincide with that intended. There is even a joke: LISP – "Lots of Idiotic Silly Parentheses". Logical errors of this kind are very difficult to find, and the poor readability of function definitions doubles the difficulty of this task.

One of the main disadvantages of the **LISP** language has traditionally been considered to be the relatively low speed of program execution. However, with the advent of powerful **LISP** machines and the development of effective **LISP** compilers, the speed of program execution has increased significantly. The relative inconvenience in mastering the **LISP** language is that there are many dialects and, unfortunately, none of them has been adopted as a standard. However, there is now hope that **Common Lisp** will become such a standard.

The LISP language is included today in the software of almost all computers produced abroad. Large IBM computers are supplied with interpreters and compilers from the **InterLISP** and **Common LISP** dialects, DEC computers - with the **Franz LISP**, **InterLISP, XLISP** systems. On personal computers, the most widespread systems are **Golden Common LISP, muLISP, IQ-LISP**.

If we talk about the global trend in the development of the **LISP** language ideology itself, it is obvious that it is connected with the creation of object-oriented versions of the language as the most suitable for the implementation ***of artificial intelligence systems***.

I would like to note one more section of computer science in which the **LISP** language is widely used.

Computer algebra is that part of computer science that deals with the development, analysis, implementation, and application of algebraic algorithms.

Algebraic algorithms differ from other algorithms by the presence of simple formal descriptions, the existence of proofs of correctness and asymptotic bounds on the execution time, which can be obtained on the basis of a well-developed mathematical theory. In addition, algebraic objects can be accurately represented in the memory of a computer, due to which algebraic transformations can be performed without loss of accuracy and significance. Algebraic algorithms are usually implemented in software systems that allow input and output of information in symbolic algebraic notations.

The choice of **LISP** as ***the language for implementing computer algebra algorithms*** is natural. This is due, firstly, to the need for dynamic memory management in connection with the problem of swelling intermediate expressions. Secondly, recursivity and list structure are natural tools for automating work with mathematical objects and operations [5, p.282].

Next, we will consider **muLISP** - one of the most successful dialects of the **LISP** language, created by Soft Warehouse Inc. (USA) [6]. Note that there are several implementations: **muLISP-81**, **muLISP-83**, **muLISP-85, muLISP-87**, **muLISP-90**.

muLISP-87 is a system with relatively few built-in functions (about 400), but it is very compact and provides creation of effective programs. It has a built-in text editor (though not very successful), an interface with Assembler, and the source code of the Flavors object-oriented programming system is supplied. The system debugging tools are more than modest. Some original solutions and new functions of the system allow creating very effective programs, although they do not fit into the classical principles of the **LISP** language. The system is supplied with very complete and well-written documentation. Unfortunately, the **muLISP** package is not compatible with the **Common LISP** standard in syntax, even if the **Common LISP** function library supplied by the company is included in it. This is a serious drawback at present. However, **muLISP** is a very powerful and convenient package that allows you to effectively solve not only artificial intelligence problems, but also problems that are not traditional for the **LISP** language, for example, it allows you to develop your own window system, organize exchange with C++ programs, etc.

The muLISP-90 programming system is a “little LISP” that runs on an IBM PC (or HP 95LX palmtop) using the MS-DOS operating system version 2.1 or later.

Of course, the **muLISP-90** system is not **Common Lisp**, although **muLISP does have a Common Lisp** compatibility package containing over 450 special **Common Lisp** forms, macros, functions, and control variables.

The system includes a screen editor, debugger, window system, interpreter and compiler. Among the numerous sample programs is DOCTOR (an "Eliza-like" program). The run-time system allows the creation of small EXE or COM files. It uses a compact internal code representation, which minimizes memory and increases execution speed. The kernel takes up only 50K.

Next, we will look at learning functional programming using the **muLISP-81, muLISP-83** and **muLISP-85** dialects.

---

(1) Explanatory Dictionary of Computing Systems / Ed. by V. Illingworth, E. L. Glazer, I. K. Pyle. - Moscow: Mashinostroenie, 1989. - 568 p.
(2) Barendregt H. Lambda calculus. Its syntax and semantics: Trans. from English. - Moscow: Mir, 1985. - 606 p.
(3) Hyvönen E., Seppänen J. The World of Lisp. In 2 volumes. Vol. 2: Programming methods and systems. - Moscow: Mir, 1990. - 319 p.
(4) McCarthy J., Abrahams PW, Edwards J., Hart TP, Levin MI LISP 1.5 Programmer's Manual. - MIT Press, Cambridge, Massachusetts, 1965.
(5) Computer Algebra: Symbolic and Algebraic Computations: Trans. from English. - Moscow: Mir, 1986. - 392 p.
(6) muLisp-85. - Reference Manual. - Software House, Inc., 1985. - 137 p.

The next step will look at the simplest data types.

Step 2.
The simplest data types. Atoms

In this step we will start to study the simplest data types, in particular we will talk about ***atoms***.

When considering data types, some initial types are first defined, which are called ***the simplest (primitive, basic, elementary) data types***.

Take a look at the diagram illustrating the classification of the simplest data types in the **LISP** language:

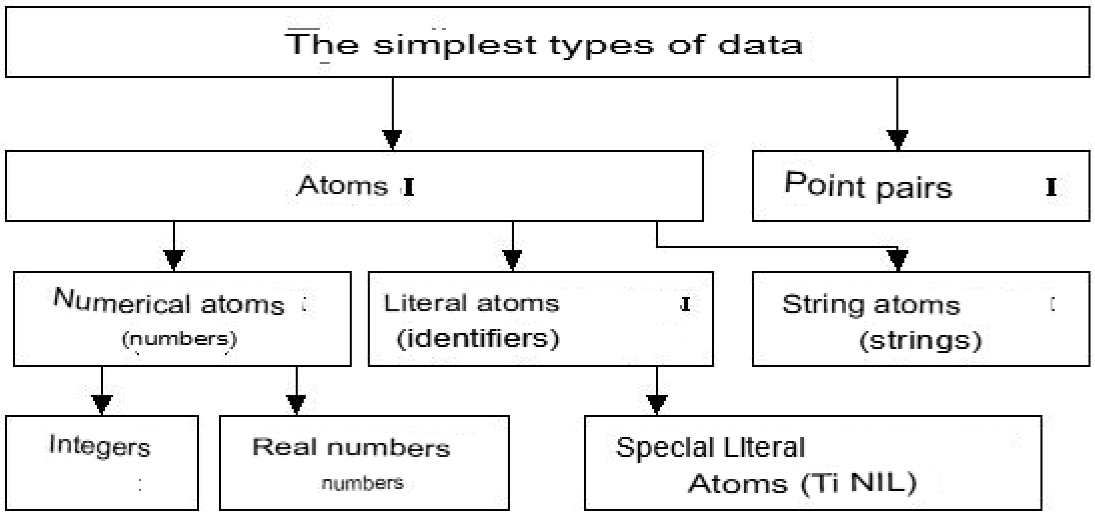


Fig. 1. Classification of the simplest data types

Let's start by talking about the contents of each block of the diagram. We'll immediately note that the **muLISP-81** and **muLISP-83** dialects lack real number atoms.

Let us explain the notations used here. The sign "::=" means "is defined as", and "|" means "or". The names of objects being defined or already defined are in angle brackets.

An atom is the most basic simple data type. In the modern world, when we study the constituent parts of an atom (in physics), such as electrons, protons, neutrons, it is easy to forget that the original meaning of the word "atom" was "that which cannot be further divided", but this is the sense in which "atom" is used in the language **LISP**. (The Greek roots "a" - not, "tom" - part; a tom also means a part of a large book.)

Atoms can be ***numeric, literal, or string***.

In Backus-Naur notation it looks like this:

```
<Atom> ::= <Numeric_atom> | <Literal_atom> |
           <String_atom> | T | NIL
```

a) A literal atom (identifier) is a sequence of letters and numbers beginning with the letter.

Note an important feature of **the muLISP-81** and **muLISP-83** interpreters: uppercase and lowercase letters are considered different, so, for example, the atoms **NAME, Name** and **name** are not identical. Examples of literal atoms: **LISP, APRPARAPR**.

Literal atoms include ***special literal atoms***:

- **NIL** is a logical value of "false" and at the same time a designation of an empty list,
- **T** is the logical value "true".

In Backus-Naur notation it looks like this:

```
<Literal_atom> ::= <Letter> |
                   <Letter> <Sequence>
<Sequence> ::= <Correct_symbol> |
               <Correct_symbol> <Sequence>
<Correct_symbol> ::= <Letter> | <Number>
```

```
  <Digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
  <Character> ::= A | B | C | D | E | F | G | H | I | G | K |
               L | M | N | O | P | Q | R | S | T | U | V |
                W | X | Y | Z |
              a | b | c | d | e | f | g | h | i | j | k |
               l | m | n | o | p | q | r | s | t | u | v |
                w | x | y | z |
```

b) Let us clarify the meaning of the term ***numeric atom*** using Backus-Naur notation for versions **muLISP-81** and **muLISP-83** (which do not have real numeric atoms):

```
  <Numeric_atom> ::= <Unsigned_integer> |
                     +<Unsigned_Integer> |
                     -<Unsigned_Integer>
  <Unsigned_Integer> ::= <Digit> | <Digit> <Unsigned_Integer>
  <Digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

Let us give examples of numerical atoms:

```
  0
  23423
  +2345
  -345
```

c) A string atom is a sequence of characters that begins and ends with quotation marks ("). String atoms (strings) differ from literal atoms (identifiers) in that they can contain atom separators as elements: spaces, periods, commas, etc., but must not contain the " (quotation mark) character inside them.

In Backus-Naur notation it looks like this:

```
  <String_atom> ::= "<A sequence of any characters,
                       not containing a quotation mark>"
```

The maximum number of characters in a string atom in the **muLISP-81** dialect is 120.

The next step will continue our consideration of the simplest data types.

Step 3.
The simplest data types. Dotted pair

In this step we will continue to study the simplest data types, in particular, we will introduce such a concept as ***a dotted pair***.

Syntactically, ***a dotted pair*** is defined as follows:

```
 <Dotted pair> ::=
        (<Atom> . <Atom>) | (<Atom> . <Dotted Pair>) |
        (<Dotted pair> . <Atom>) |
        (<Dotted pair> . <Dotted pair>)
```

Thus, ***a dotted pair*** is an ordered pair whose elements can be either atoms or dotted pairs, and the dot acts as a separator. Here are some examples of dotted pairs:

```
  (JOHN . SMITH)
  (SIN . (X . (PLUS . Y)))
```

```
   (A . ((B . 1) . NIL))
```

It is often convenient not to distinguish between atoms and dotted pairs. In this case, we speak of ***S-expressions*** (from Symbolic). Thus, by definition

```
   *****************************************************
   * S - expression ::= Atom ¦ Dotted pair *
   *****************************************************
```

or else

```
     S-expression ::= Atom | (S-expression. S-expression)
```

"An S-expression is either an atom or a left parenthesis followed by an S-expression, a period, an S-expression, and a right parenthesis."

---

*Notes. 1. Let us construct **a grammar** for S-expressions of the **LISP** language. If an atom is considered a terminal symbol, then the grammar for S-expressions looks like this:*

```
   <S> --> ATOM
   <S> --> (<S>.<S>)
```

Let's give an example of output in this context-free grammar:

```
   <S> --> (<S>.<S>)                           -->
       --> ((<S>.<S>).<S>)                     -->
       --> ((ATOM . <S>) . <S>)                -->
       --> ((ATOM . (<S> . <S>)) . <S>)        -->
       --> ((ATOM . (ATOM . ATOM)) . <S>)      -->
       --> ((ATOM . (ATOM . ATOM))) . ATOM)
```

*2. Numbers in the **muLISP-85** system can be divided into:*

- ***whole** and*
- ***small**.*

*In the internal representation, **integers** are further subdivided into:*

- ***small integers** (less than 65536 in magnitude) and*
- ***large whole ones**.*

*With the exception of the **EQ** function, the distinction between small and large integers does not affect other **muLISP** functions.*

***Fractional** and **large integer numbers** are not stored in a unique way, so numbers with the same values can coexist in the system.*

***A number** is described using four pointer elements:*

- ***identity**. This element contains a pointer to the number itself. Therefore, numbers should not be referenced, since they express themselves. This element can be accessed as a **CAR** element of the number, but it cannot be modified by **muLISP** ;*
- ***sign (sign)**. This element specifies the sign of the number and whether it is a small integer, large integer, or fractional number. The element contains a pointer to one of six symbols, according to the table:*

Table 1. **Values of the element "sign"**

	Positive	**Negative**
Small number	**NIL**	**T**

Large number	**LAMBDA**	**NLAMBDA**
Fractional number	**MACRO**	**SPECIAL**

*The contents of the sign element can be accessed as **the cdr** element of the number, but cannot be modified by the **muLISP** program ;*

- ***length (length)***:
  - *if the number is a small integer, then this element contains the magnitude of the integer;*
  - *if the number is a large integer, then this element contains the word length of the number vector;*
  - *if the number is a fractional number, then this element contains a pointer to the numerator of the fractional number, which must be an integer;*
- ***vector (vector)***:
  - *if the number is a small integer, then this element contains a pointer to another small integer or 1;*
  - *if the number is a large integer, then this element contains a pointer to the least significant byte (word) of the number vector.*

***A number vector** is a series of contiguous bytes (words) in memory used to store the magnitude of a **muLISP** number in binary form. The bytes (words) of a number vector are stored in big-endian order (more significant bytes (words) are stored sequentially, starting with lower memory addresses). If the number is a fractional number, this element contains a pointer to the denominator of the number, which must be a positive integer.*

In the next step the concept of a list will be introduced.

Step 4.
List as a fundamental data type

Here we introduce the concept of a list and establish its connection with an atom and a dotted pair.

The main fundamental data type in the **LISP** language is a list. The name of the **LISP** language is derived from " **LIS** t **P** rocessing".

The list is recursively defined as:

```
      List ::= NIL | (S-expression. List)
```

By this definition, any non-empty list is a dotted pair.

Here are ***some examples of lists***:

```
  (LISP Functional language)
  (5 6 (8 (9)) 23 NIL)
```

It is important to highlight the concept of ***an empty list***, which contains no elements, it is denoted by **()** or the atom **NIL**. For example, **(NIL ())** is a list consisting of two empty lists.

---

*Note: One of the paradoxes of the **LISP** language is that the **empty** list () is the **NIL** **atom** !*

Some say that this was originally done because of the specifics of the machine instruction system of the very old 709 computer. Others are sure that such an identity has always been considered convenient from a programmer's point of view. No one knows the exact reason.

Another representation of lists is often used in literature - ***graphical notation***. It conveys the structure of lists more clearly. In graphical notation, it is easy, for example, to depict a cyclic list that can be organized using the **LISP** language, but it is impossible to depict it in symbolic notation! Graphic notation relies entirely on the reference structure of **LISP** lists.

The RAM of the computer on which **LISP** "runs" is logically divided into small areas called ***list cells***.

A list cell consists of fields named **CAR** and **CDR**, each of which contains a pointer. The pointer may refer to another list cell or to another **LISP** object, such as an atom.

Pointers between cells form a kind of chain, along which one can get from the previous cell to the next one and so on, finally, to atomic objects. Each atom known to the system is written in a certain place in memory only ***once***.

Graphically, we will represent a list cell as a rectangle divided into the **CAR** and **CDR** fields. The pointer is depicted as an arrow starting in one of the parts of the rectangle and ending at the image of another cell or atom to which the pointer refers.

Example 1. Let's represent the list (A B C D) in graphical notation:

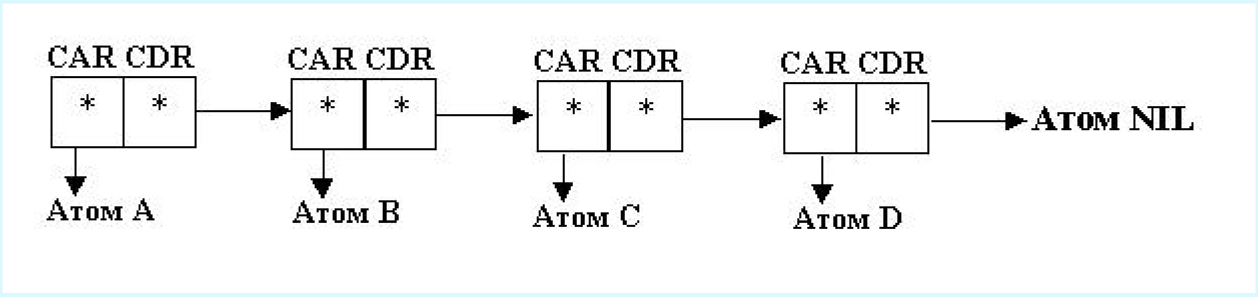


Fig. 1. List (A B C D)

Example 2. Let's represent the list (A (16 (A 25)) (B) (21)) ***in graphical notation***:

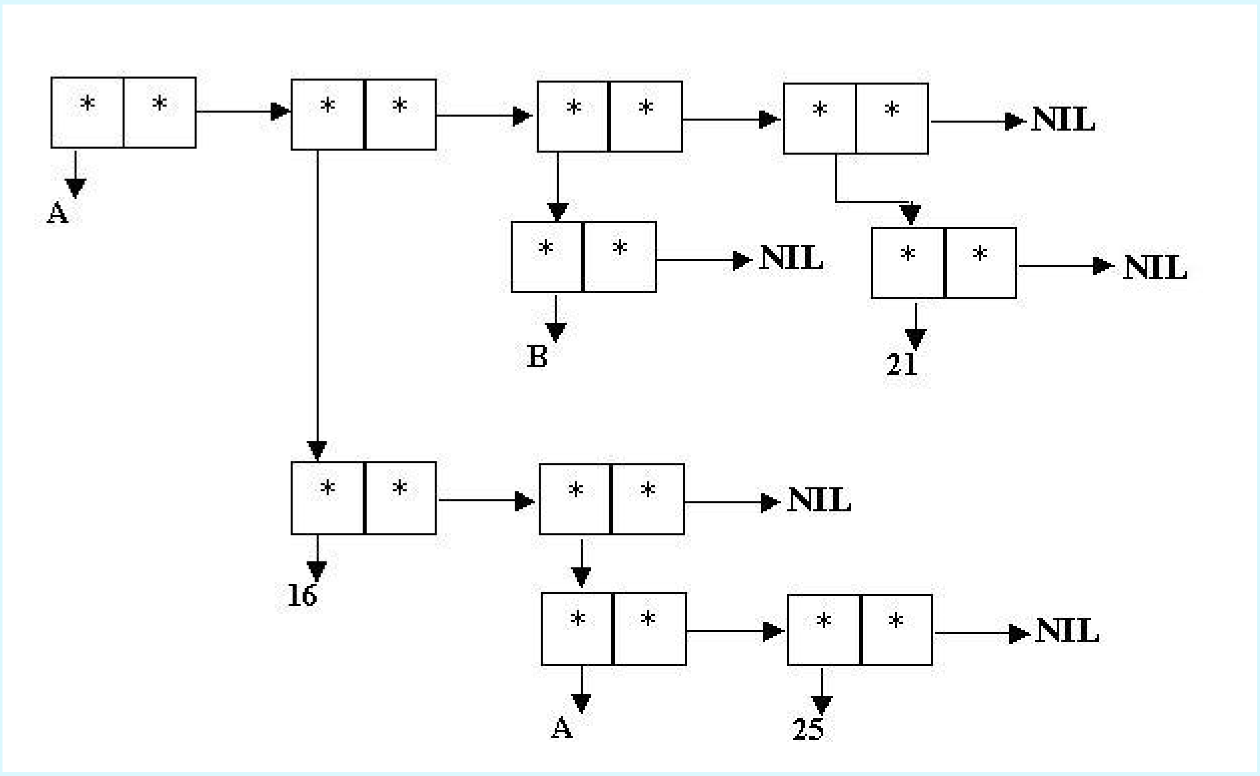


Fig.2. List (A (16 (A 25)) (B) (21))

Example 3. The structure, depicted ***in graphical notation*** as

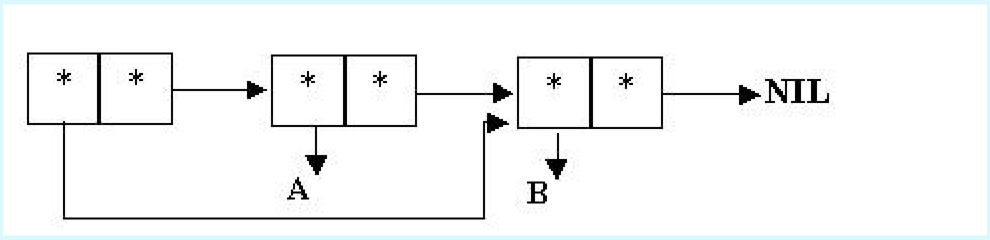


Fig.3. Example of structure

corresponds to the expression in list notation: ((B) A B).

In this respect it is completely equivalent to the structure shown below (Fig. 4):

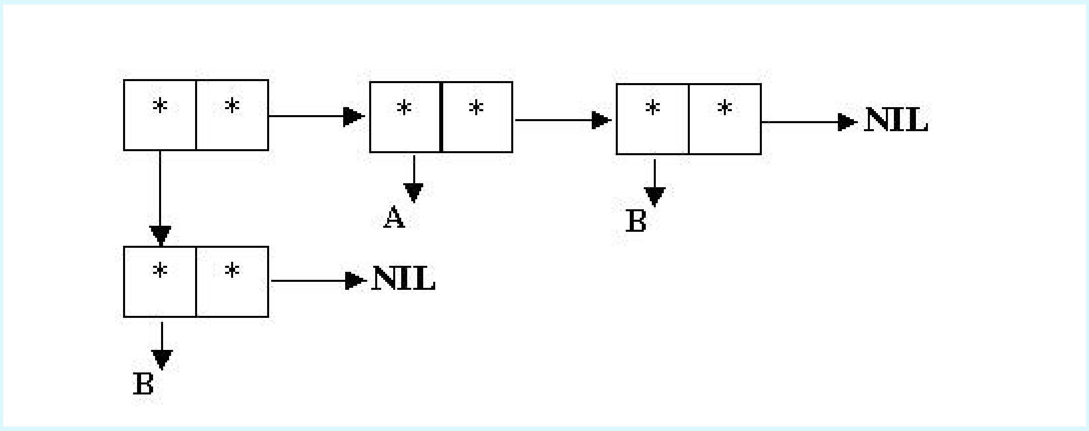


Fig.4. Example of equivalent structure

Note the different number of Lisp cells!

Now we will talk about the graphical representation ***of point pairs***.

When a list is syntactically represented using ***list notation*** (a sequence of elements enclosed in parentheses), the terminating **CDR** pointer is assumed to point to the **NIL** atom, e.g. the list

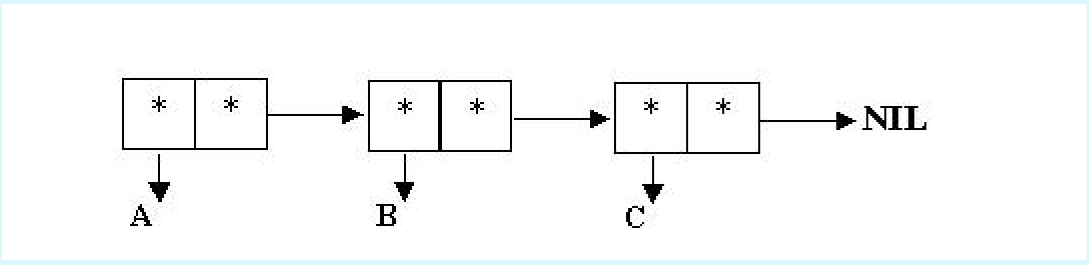


Fig.5. List (A B C)

is written as (A B C).

Sometimes it is desirable to be able to include in the last element of a list a **CDR** pointer that points to an atom other than **NIL**. In this case, another notation called ***dot notation*** can be used, in which each element of the list is written as a pair of subelements representing **the CAR** and **CDR** fields of the element. The subelements are enclosed in parentheses and separated from each other ***by a period***.

The following example uses a representation of point pairs as ***binary trees with labeled leaves*** in addition to a graphical representation of point pairs.

Example 4. A structure whose graphical representation looks like this:

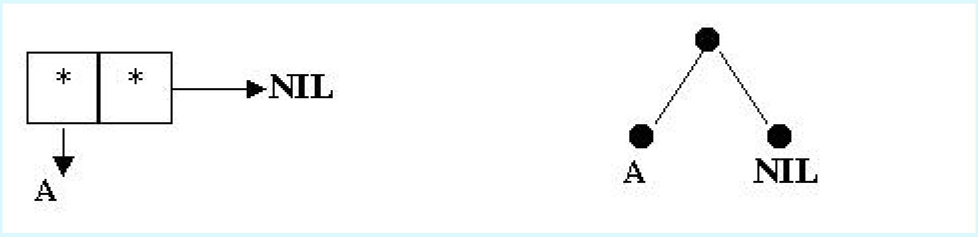


Fig.6. Structure and graphical representation

is written in dot notation as **(A . NIL)**.

The structure with the graphical representation

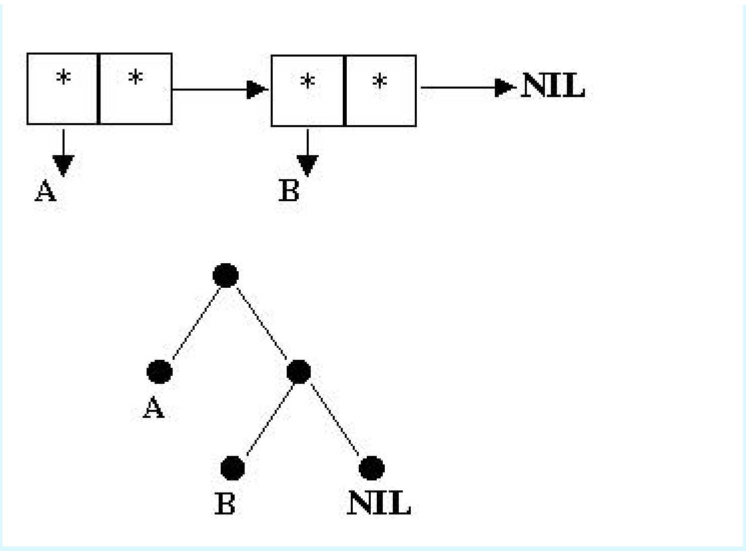


Fig.7. Structure and graphical representation

is written as **(A . (B . NIL))** in dot notation.

Now the element with a graphical representation

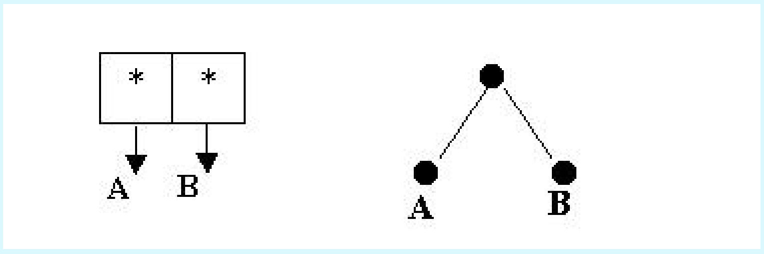


Fig.8. Structure and graphical representation

which could not be represented in list notation is written using dot notation as **(A. B)**.

The structure with the graphical representation

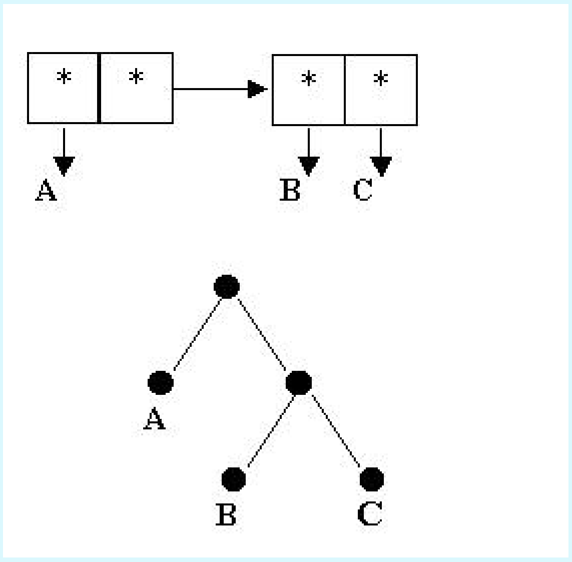


Fig.9. Structure and graphical representation

is written as **(A . (B . C))**.

The structure

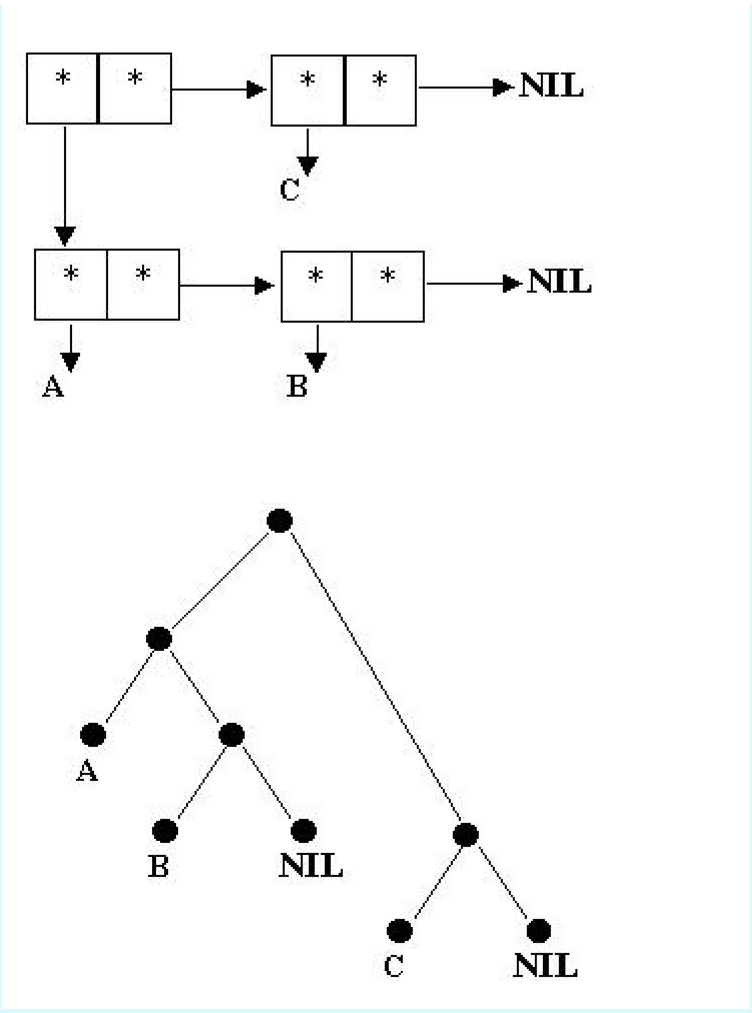


Fig.10. Structure and graphical representation

is written as **((A . (B. NIL)) . (C . NIL))**.

The structure

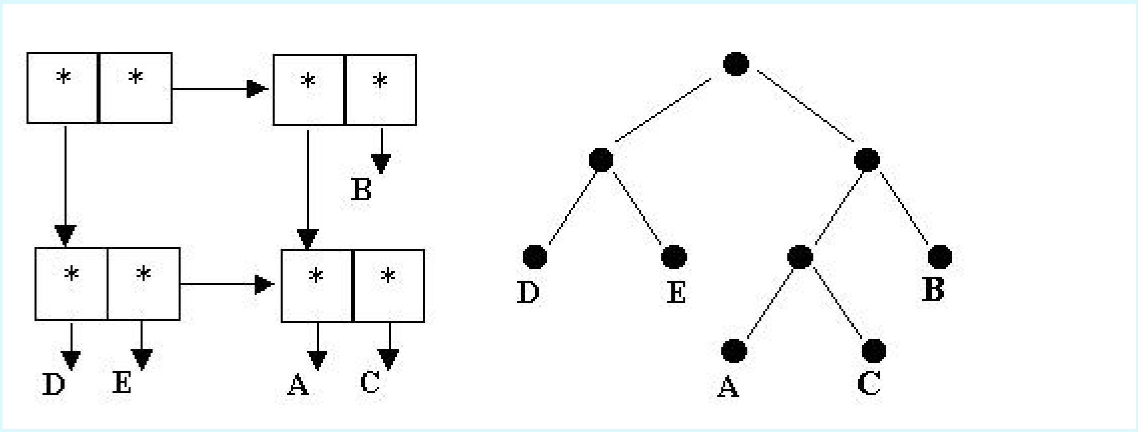


Fig.11. Structure and graphical representation

is written as **((D . E) . ((A . C) . B))**.

The dotted notation of lists has some advantages over the list notation. It is more general, since any list can be rewritten in dotted notations, but even the simplest dotted expression **(A . B)** cannot be represented by list notation. When an expression is built from a small, and most importantly, pre-known number of elements, dotted constructions are preferable. If the number of elements is relatively large and can be variable, then it is more appropriate to use list constructions.

There is a rule that allows you to simplify the notation of dot expressions. To return from ***dot notation to list notation***, when such a return is possible (!), do the following: if a period is before an opening parenthesis, then replace it with a space, throwing out both this opening parenthesis and its paired closing parenthesis. The same rule allows you to get rid of unnecessary **NILs**, if you remember that **NIL** is equivalent to a pair of adjacent parentheses (). We will call the resulting dot notation ***abbreviated dot notation***.

Note that this rule can be applied to any well-formed S-expression, even if it does not ultimately reduce to a list.

Let's give some examples:

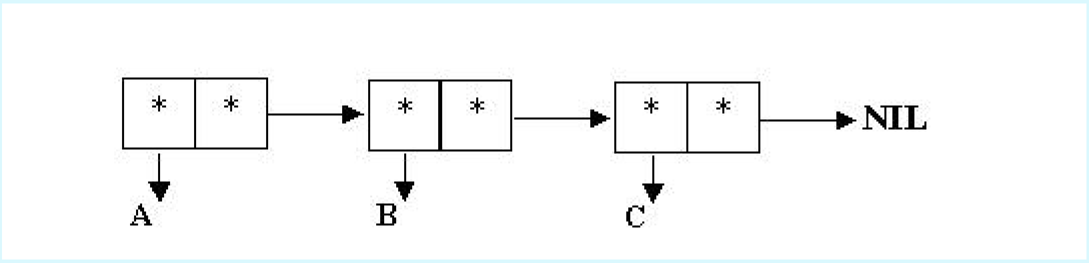
```
  the entry (A . (B . C)) is equivalent to (A B . C),
  the notation ((A . B) . (C . D)) is equivalent to ((A . B) C . D).
```

It is easy to verify that the full dot notation can be reconstructed from the abbreviated dot notation. To do this, each space is replaced by a period, followed by an opening parenthesis. The corresponding closing parenthesis is inserted immediately before the closest closing parenthesis to the right of this space, which does not have a matching opening parenthesis also to the right of the space.

Dot notation allows us to expand the class of objects that can be represented using list notation. The situation can be compared to the use of fractions in arithmetic or complex numbers in mathematical analysis.

By using dot notation, you can reduce the amount of memory required. For example, the data structure **(A B C)**, represented as a list entry, requires three cells, although the same data can be represented as **(A B . C)**, which requires only two cells. A more compact representation can reduce the amount of computation by requiring fewer memory accesses.

In accordance with the above, at least two groups of cells must be allocated in the computer's memory - a group of atomic information cells and a group of cells corresponding to dot pairs. These groups are indeed allocated - usually in the form of connected memory areas. The area in which the atomic information cells are located is traditionally called ***the object list***, and the area in which the pairs are located is called ***the list memory area***. Both names are not entirely successful. The object list is not a list in the sense in which this term is used in the language. The cells of the list memory area can be linked not only in chains representing lists, according to the scheme



but also in other structures. Moreover, structures that do not allow description by linguistic means are possible. The simplest example is shown below:

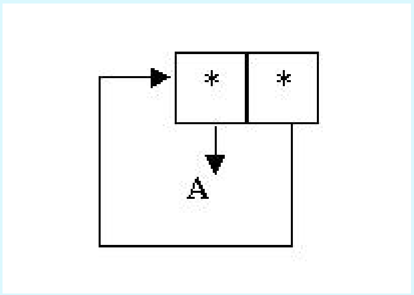


Fig.12. Example of structure

In the next step we will begin to get acquainted with the basic functions of the **LISP** language.

Step 5. Arithmetic functions of the LISP language

In this step we will list the basic arithmetic functions of the LISP language.

To program with functions, one must be able to define a sufficiently rich set of basic functions, and then use composition to be able to define new functions in terms of the original ones. The question arises whether anything more is needed than a set of original functions and the ability to compose them for programming. Our answer is that nothing more is needed, and the purpose of this section is to illustrate and confirm this point.

LISP uses Cambridge Polish notation to write functions (the function name is written first, followed by the sequence of its arguments), with brackets placed wherever they make sense. This allows for a simple and uniform syntax, but brackets often make structures less visual.

---

*Note: The arithmetic functions perform basic mathematical operations on integers and fractional numbers. In **muLISP-85,** the user can choose to work with either exact or approximate rational arithmetic. For exact rational arithmetic, the size of integers, numerators, and denominators is limited to approximately 225,000 decimal places. **For approximate rational arithmetic (usually called floating-point arithmetic),** the precision used is also set within these limits.*

If an arithmetic function that requires integer arguments is called with non-integer arguments, a "Non-integer argument" error occurs.

If an arithmetic function that requires numeric arguments is called with non-numeric arguments, a "Non-numeric argument" error interrupt occurs.

If a numeric function is called with insufficient arguments, an "Insufficient arguments" error occurs.

If an attempt is made to divide by zero, a "Division by zero" error interrupt occurs.

Once an interrupt occurs, the console displays a diagnostic message, a function call, and a prompt in the form of interrupt options. The interrupt allows the user to determine the cause of the error and the type of action.

Although fractional numbers can be written either in decimal notation or as a fraction (using a slash), they are stored internally as fractional numbers consisting of two integers.

Note that when entering fractional numbers, spaces are not placed near the period or slash, and at least one decimal number must be placed before the period.

*The control variables ***READ-BASE***, ***PRINT-BASE***, and ***PRINT-POINT*** specify how numbers are represented for input and output.*

The following table lists the basic arithmetic functions of the **LISP language in the muLISP-85** dialect.

Table 1. **Basic arithmetic functions of the LISP language**

Function	**Purpose**

(+ N1 N2 ... Nm)	Returns the sum of numbers **N1, ..., Nm**.
(- N1 N2 ... Nm)	Returns the difference between **N1** and the sum of numbers **N2, ..., Nm**.
(* N1 N2 ... Nm)	Returns the product of numbers **N1, ..., Nm**.
(/ N1 N2 ... Nm)	Returns the result of dividing **N1** by the product of numbers **N2, ..., Nm**.
(MOD N M)	Returns the absolute value **of N** divided by **M.**
(GCD N1 N2 ... Nm)	Returns the greatest common divisor of numbers **N1** through **Nm**.
(LCM N1 N2 ... Nm)	Returns the least common multiple of numbers **N1** through **Nm**.
(NUMERATOR N)	Returns the numerator of the number **N**.
(DENOMINATOR N)	Returns the denominator of the number **N**.
(LOGAND N1 N2 ... Nm)	Returns the result of performing a bitwise logical "AND" on integers **N1, ..., Nm**.
(LOGIOR N1 N2 ... Nm)	Returns the result of performing a bitwise logical "OR" on integers **N1, N2, ..., Nm**.
(LOGXOR N1 N2 ... Nm)	Returns the result of performing a bitwise logical exclusive OR (**XOR**) on integers **N1, N2, ..., Nm**.
(LOGNOT N)	Returns the result of performing a bitwise logical NOT on a number **N**.
(SHIFT N M)	Returns the result of shifting **N** left by **M** bits.
(BITLENGTH N)	Returns the number of bits required to hold the number **N**.
(MAX N1 N2 ... Nm)	Returns the largest argument.
(MIN N1 N2 ... Nm)	Returns the smallest argument.
(ADD1 N)	Returns **(+ N 1)**.
(SUB1 N)	Returns **(- N 1)**.
(INCQ SYMBOL)	Increments the value of **SYMBOL** and returns the new value.
(DECQ SYMBOL)	Decrements the value of **SYMBOL** and returns the new value.
(ABS N)	Returns the absolute value **of N**.
(SIGN N)	Returns the sign of the number **N**.
(REM N M)	Returns the remainder when **N** is divided by **M**.
(DIVIDE N M)	Returns a dotted pair representing the quotient and remainder of the integer division **of N** by **M**.
(PRECISION N)	Takes **N** as the precision value and returns the previous precision value.
(UNDERFLOW N)	Changing the calculation precision value.
(FLOOR ...)	“Truncating” the value at the lower limit.
(CEILING ...)	“Truncating” the value at the upper limit.
(TRUNCATE ...)	“Truncating” a value by bringing it closer to zero.
(ROUND...)	Rounding off the value.
(QUOTE <argument>)	Prevents the evaluation of the value of its argument and returns the argument itself.

1. The function **(+ N1 N2 ... Nm)** returns the sum of the numbers N1, ..., Nm. If the function is called without arguments, it returns 0. For example:

```
    $ (+ 2 3 4)        $ (+ 0.25)
    9                  0.25
    $ (+ -5 3)         $ (+ 10 5 3)
    -2                 18
    $ (+ 1/6 0.7)      $ (+)
    0.8666666          0
```

2. The function **(- N1 N2 ... Nm)** returns the difference between N1 and the sum of the numbers N2, ..., Nm. The function **(- N)** returns -N. If the function is called without arguments, an interrupt occurs with the error "Not enough arguments". For example:

```
    $ (- 12 5)          $ (- 17/9 1.5)
    7                   0.3888888
    $ (- 12 5 -3)       $ (- 8)
    10                  -8
```

3. The function **(* N1 N2 ... Nm)** returns the product of numbers N1, ..., Nm. If the function is called without arguments, it returns 1. For example:

```
    $ (* 3 4 5)         $ (*)
    60                  1
    $ (* 5/6 -0.7)      $ (* 2/5)
    -0.5833333          0.4
```

4. The function **(/ N1 N2 ... Nm)** returns the result of dividing N1 by the product of numbers with N2, ..., Nm. If the function is called with a single argument, it returns the inverse of the argument. If the function is called without arguments, an interrupt occurs with the error “Not enough arguments”. If any argument other than the first is zero, an interrupt occurs with the error “Division by zero”. For example:

```
    $ (/ 12 8)        $ (/ -4.7 1.3)
    1.5               -3.6153846
    $ (/ 12 5 -3)     $ (/ 5)
    -0.8              0.2
```

5. If **N** and **M** are numbers, then the function **(MOD N M)** returns the result of dividing **N** by **M** modulo. For example:

```
    $ (MOD 7 3) $ (MOD 7 -3)
    1               -2
    $ (MOD -7 3) $ (MOD -7 -3)
    2               -1
```

If **M** (the divisor) is zero, a “Divide by Zero” error interrupt occurs.

Note that the **MOD** function is defined so that the **FLOOR** and **MOD** functions preserve the relationship between quotients and remainders: **N = M * (FLOOR N M) + (MOD N M)**. Here is the function code:

```
    (DEFINE MOD (N M)
       ( (AND (NUMBERP N) (NUMBERP M))
           ( (ZEROP M)
                (BREAK (LIST 'MOD N M) '"Zero Divide") )
           (- N (* M (FLOOR N M))) )
       ( BREAK (LIST 'MOD N M) '"Nonnumeric Argument" )
    )
```

In mathematical terms, the function **(MOD N1 N2)** returns the smallest non-negative residue of the number **N1** modulo **N2**.

6. The function **(GCD N1 N2... Nm)** returns the greatest common divisor of numbers from **N1** to **Nm**. The function **(GCD N)** returns the absolute value of the number **N**. If the function is called without arguments, then **GCD** returns 0.

The GCD function of two numbers is the largest non-negative integer that is divisible by both numbers. **The GCD** of two fractional numbers is **the GCD** of the numerators divided by **the LCM** of the denominators. For example:

```
    $ (GCD 12 -16 20)       $ (GCD 0 0)
    4                       0
    $ (GCD 14/15 8/9)       $ (GCD)
    0.0444444               0
    $ (GCD 0 5)             $ (GCD -7)
    5                       7
```

7. The **(LCM N1 N2 ... Nm)** function returns the least common multiple of numbers **N1** through **Nm**. The **(LCM N)** function returns the absolute value of **N.** If the **LCM** function is called without arguments, an interrupt occurs with the error “Not enough arguments”. The **LCM** function of two integers returns the smallest integer that is a multiple of both given numbers. For example:

```
    $ (LCM 12 -16 20)       $ (LCM 0 5)       $ (LCM -7)
    240                     0                 7
    $ (LCM 14/15 8/9)       $ (LCM 0 0)
    18.6666666              0
```

In general, the **LCM** function of two numbers **N** and **M** (integers or fractions) can be defined using the **GCD** function as **(/ (ABS (* N M)) (GCD N M))**.

8. If the number N is a fraction, then the function **(NUMERATOR N)** returns ***the numerator*** **N**. If **N** is an integer, then the function **(NUMERATOR N)** returns **N**. The numerator is always an integer. For example:

```
    $ (NUMERATOR 10/8)     $ (NUMERATOR -8)
    5                      -8
    $ (NUMERATOR 0.75)
    3
```

9. If **N** is a fractional number, the function **(DENOMINATOR N)** returns ***the denominator of*** **N**. If **N** is an integer, **(DENOMINATOR N)** returns 1. The denominator is always a positive integer. For example:

```
    $ (DENOMINATOR 10/8)    $ (DENOMINATOR -8)
    4                       1
    $ (DENOMINATOR 0.75)
    4
```

The bitwise logic functions discussed below operate only on ***integers***. If they are called with non-integer arguments, a “Non-integer argument” error interrupt occurs.

10. The **(LOGAND N1 N2... Nm)** function returns the result of performing a bitwise logical "AND" on integers **N1, ..., Nm**. If the **LOGAND** function is called without arguments, it returns -1. For example:

```
    $ (LOGAND 11 6)
    2
```

11. The function **(LOGIOR N1 N2 ... Nm)** returns the result of performing a bitwise logical "OR" on integers **N1, N2, ..., Nm**. If the **LOGIOR** function is called without arguments, it returns 0. For example:

```
    $ (LOGIOR 11 6)
    15
```

12. The **(LOGXOR N1 N2 ... Nm)** function returns the result of performing a bitwise logical exclusive "OR" (**XOR**) on integers **N1, N2, ..., Nm**. If the **LOGXOR** function is called without arguments, it returns 0. For example:

```
    $ (LOGXOR 11 6)
    13
```

13. The **(LOGNOT N)** function returns the result of performing a bitwise logical "NOT" on the number **N**. For example:

```
$ (LOGNOT 9)
65526
```

14. If **M** is a positive number, the **(SHIFT N M)** function returns the result of shifting **N** to the left by **M** bits. If **M** is a negative number, **N** is "shifted" to the right by **-M** bits. For example:

```
$ (SHIFT 5 2)      $ (SHIFT 5 -2)
20                 1
$ (SHIFT 6 3)
48% 48=110000b
```

15. The **(BITLENGTH N)** function returns the number of bits required to hold the number **N**. For example:

```
$ (BITLENGTH 4)    $ (BITLENGTH 8)
3                  4
$ (BITLENGTH 7)    $ (BITLENGTH 48)
3 6 % 48 = 110000b
```

The following elementary functions are defined in the **muLISP-85** COMMON.LSP **file: EXP, EXPT, LOG, SQRT, ISQRT, SIN, COS, TAN, ASIN, ACOS, ATAN, PI, RANDOM**.

Let us recall that ***elementary functions*** are a class of functions consisting of power functions, polynomials, rational functions, exponential functions, logarithmic functions, trigonometric functions, inverse trigonometric functions, hyperbolic functions, inverse hyperbolic functions, as well as functions obtained from the above using the four arithmetic operations (addition, subtraction, multiplication, and division) and superpositions applied a finite number of times.

16. The function **(MAX N1 N2 ... Nm)** returns its largest argument. For example:

```
$ (MAX 5 -7 4)     $ (MAX -4)
5                  -4
$ (MAX 2/3 0.6)
0.6666666
```

If the **MAX** function is called without arguments, an interrupt occurs with the error "Not enough arguments". The function code is as follows:

```
(DEFUN MAX LST
   ( (NULL LST)
      ((BREAK (LIST 'MAX) '"Insufficient Arguments") )
   ( (NULL (CDR LST)) (CAR LST) )
   ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
      ( (< (CAR LST) (CADR LST))
          (APPLY 'MAX (CONS (CADR LST) (CDDR LST))) )
      (APPLY 'MAX (CONS (CAR LST) (CDDR LST))) )
   (APPLY 'MAX
          (CONS (BREAK (LIST 'MAX (CAR LST) (CADR LST))
                              '"Nonnumeric Argument")
                      (CDDR LST)))
)
```

17. The function **(MIN N1 N2 ... Nm)** returns its smallest argument. For example:

```
$ (MIN 5 -7 4)       $ (MIN -4)
-7                   -4
$ (MIN 2/3 0.6)
0.6
```

If **MIN** is called without arguments, an “Insufficient arguments” error occurs. The function code is:

```
(DEFUN MIN LST
   ( (NULL LST)
       (BREAK (LIST 'MIN) '"Insufficient Arguments") )
   ( (NULL (CDR LST)) (CAR LST) )
   ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
       ((< CAR LST) (CADR LST))
          (APPLY 'MIN (CONS (CAR LST) (CDDR LST))) )
   (APPLY 'MIN (CONS (BREAK
                        (LIST 'MIN (CAR LST) (CADR LST))
                        '"Nonnumeric Argument")
                  (CDDR LST)))
)
```

18. The function **(ADD1 N)** returns **(+ N 1)**. For example:

```
$ (ADD1 3)        $ (ADD1 -3)
4                 -2
$ (ADD 2/3)
1.6666666
```

If **ADD1** is called without arguments, an interrupt occurs with the error “Not enough arguments”. Here is the function code:

```
(DEFUN ADD1 (N)
   ( (NUMBERP N) (+ N 1))
   ( BREAK (LIST 'ADD1 N) '"Nonnumeric Argument")
)
```

19. The function **(SUB1 N)** returns **(- N 1)**. For example:

```
$ (SUB1 3)       $ (SUB1 -3)
2                -4
$ (SUB1 0.25)
-0.75
```

If the **SUB1** function is called without arguments, an interrupt occurs with the error "Not enough arguments". Here is the function code:

```
(DEFUN SUB1 (N)
   ( (NUMBERP N) (- N 1) )
   ( BREAK (LIST 'SUB1 N) '"Nonnumeric Argument" )
)
```

20. The **(INCQ SYMBOL) function *increments*** the value of **SYMBOL** and returns the new value. If **N** is a number, the **(INCQ SYMBOL N) function *adds*** **N** to the value of **SYMBOL** and returns the new value. For example (**SETQ** is the assignment function):

```
$ (SETQ CTR 5)   $ (INCQ CTR 2)
5                8
$ (INCQ CTR)     $ CTR
6                8
$ CTR
6
```

If **SYMBOL** is not a **muLISP** symbol, the function generates a “Non-symbolic argument” error trap.

21. The **(DECQ SYMBOL) function *decrements*** the value of **SYMBOL** and returns the new value. If **N** is a number, the **(DECQ SYMBOL N) function *subtracts*** **N** from the value of **SYMBOL** and returns the new value. For example:

```
$ (SETQ CTR 5)    $ (DECQ CTR 2)
5                 2
$ (DECQ CTR)      $ CTR
4                 2
$ CTR
4
```

If **SYMBOL** is not a **muLISP** symbol, the **DECQ** function generates a “Non-symbolic argument” error trap.

22. If **N** is a number, then the function **(ABS N)** returns the absolute value **of N.** For example:

```
$ (ABS 4/6)       $ (ABS 0)
0.6666666         0
$ (ABS -3)
3
```

Code functions simple:

```
(DEFUN ABS (N)
   ( (NUMBERP N) ( (MINUSP N) (- N) ) N )
   ( BREAK (LIST 'ABS N) '"Nonnumeric Argument" )
)
```

23. If **N** is a number, then the **(SIGNUM N)** function returns 1, -1, or 0 depending on whether **N** is positive, negative, or zero. For example:

```
$ (SIGNUM 7/2)
1
$ (SIGNUM 0)
0
$ (SIGNUM -0.2)
-1
```

Here is the function code:

```
(DELETE SIGNUP (N)
   ( (NUMBERP N) ((ZEROP N) 0) ((PLUSP N) 1) -1 )
   ( BREAK (LIST 'SIGNUM N) '"Nonnumeric Argument" )
)
```

24. If **N** and **M** are numbers, then the function **(REM N M)** returns the remainder of the division **of N** by **M.** If **M** (the divisor) is zero, then a "Division by zero" error interrupt occurs.

The REM function is defined so that the **TRUNCATE** and **REM** functions preserve the relationship between quotients and remainders: **N = M * (TRUNCATE N M) + (REM N M)**. For example:

```
$(REM 7 3) $(REM 7 -3)
1           1
$(REM -7 3) $(REM -7 -3)
-1          -1
```

Here is the function code:

```
(DEFINE REM (N M)
   ( (AND (NUMBERP N) (NUMBERP M))
       ( (ZEROP M)
           (BREAK (LIST 'REM N M) '"Zero Divide") )
           (- N (* M (TRUNCATE N M))) )
   ( BREAK (LIST 'REM N M) '"Nonnumeric Argument" )
)
```

25. If **N** is a number, then the **(DIVIDE N)** function returns a dotted pair whose **CAR** element is **(TRUNCATE N)** and whose **CDR** element is **(REM N)**.

If **N** and **M** are numbers, then the function **(DIVIDE N M)** returns a dotted pair whose **CAR** element is **(TRUNCATE N M)** and whose **CDR** element is **(REM N M).**

If **M** (the divisor) is zero, a "Division by zero" error interrupt occurs.

In cases where the truncated quotient and remainder are required simultaneously, it is more advantageous to use the **DIVIDE** function than to calculate them separately. For example:

```
   $ (DIVIDE 7 3)      $ (DIVIDE 7 -3)
   (2. 1)             (-2. 1)
   $ (DIVIDE -7 3)     $ (DIVIDE -7 -3)
   (-2. -1)           (2. -1)
```

Here is the function code:

```
   (DEFINE DIVIDE (N M)
      ( (AND (NUMBERP N) (NUMBERP M))
         ( (ZEROP M)
            (BREAK (LIST 'DIVIDE N M) '"Zero Divide") )
         (CONS (TRUNCATE N M) (REM N M)) )
      ( BREAK (LIST 'DIVIDE N M) '"Nonnumeric Argument")
   )
```

26. If **N** is a positive integer, then the function **(PRECISION N)** takes **N** as the precision value and returns the previous precision value.

If **N** is zero, the **(PRECISION N)** function takes the precision value as infinity and returns the previous value.

If **N** is absent or is neither zero nor a positive integer, the **(PRECISION N)** function returns the current precision value without changing it.

If the precision value is set to a positive integer **N, then a new fractional number generated by numeric functions is automatically rounded if more than N** memory words are required to accommodate the smaller of its numerator and denominator (a memory word consists of 16 bits).

Such a fractional number is replaced by an approximation, one that requires no more than **N** words to accommodate the smallest of the numerator and denominator values.

The initial value of precision is assumed to be 1. For example:

```
   $ (PRECISION 0)
   1
   $ (PRECISION 5)
   0
   $ (PRECISION)
   5
   $ (PRECISION 1)
   5
```

Thus, the **PI** function defined in the **COMMON.LSP** file can be approximated to any desired degree of accuracy in **muLISP**. Here is an example of calculating **PI** to 100 digits:

```
  $ (PROGN (PRECISION 10) (SETQ *PRINT-POINT* 100) (PI))
  3.14159265358979323846264338327950288419716939937510582097
  4459230781640628620899862803482534211706
```

27. Often in the process of calculations with approximation, numbers are required that are smaller in absolute value than some floating-point number tending to zero (**underflow** situation).

If **N** is a positive integer, the **(UNDERFLOW N)** function sets the value at which **underflow** occurs to a value of **65536**$^{-N}$ and returns the previous **underflow** value.

If **N** is zero, the **(UNDERFLOW N)** function prevents **underflow** from occurring and returns the previous value.

If **N** is absent or is neither 0 nor a positive integer, the **(UNDERFLOW N)** function returns the current value **of underflow** without changing it.

Initially, **underflow** is taken to be 7. Below is a table of 2 numerical values **of underflow** for different values **of N**:

Table 1. **Underflow values for different N values**

N	**65536**$^{-N}$
1	$1.53*10^{-5}$
2	$2.33*10^{-10}$
3	$3.55*10^{-15}$
4	$5.42*10^{-20}$
5	$8.27*10^{-25}$
6	$1.26*10^{-29}$
7	$1.93*10^{-34}$ (**underflow** by default)
8	$2.94*10^{-39}$
9	$4.48*10^{-44}$
10	$6.84*10^{-49}$

For example:

```
    $ (UNDERFLOW 0)
    7
    $ (UNDERFLOW 5)
    0
    $ (UNDERFLOW)
    5
    $ (UNDERFLOW 7)
    5
```

28. If **N** is an integer, then the **(FLOOR N)** function returns **N**. If **N** is a fractional number, then the **FLOOR** function returns the largest integer less than **N**. In other words, the **FLOOR** function “truncates” **N** at the lower bound.

If **N** and **M** are numbers, then the function **(FLOOR N M) truncates the quotient of N** divided by **M** at the lower bound. The function **(FLOOR N M)** is equivalent to **(FLOOR (/ N M))**.

If **N** (the divisor) is zero, then a "Division by zero" error interrupt occurs. For example:

```
    $ (FLOOR 7.3)       $ (FLOOR 8 3)
    7                   2
    $ (FLOOR -3.5)      $ (FLOOR -15/4 2)
    -4                  -2
```

29. If **N** is an integer, then **(CEILING N)** returns **N**. If **N** is a fraction, then **(CEILING N)** returns the smallest integer greater than **N**. In other words, **(CEILING N)** "truncates" **N** at its upper bound.

If **N** and **M** are numbers, then the function **(CEILING N M) truncates the quotient of N** divided by **M** at the upper bound. The function **(CEILING N M)** is equivalent to the function **(CEILING (/ N M))**.

If **M** (the divisor) is zero, then a "Division by zero" error interrupt occurs. For example:

```
$ (CEILING 7.3)     $ (CEILING 8 3)
8                   3
$ (CEILING -3.5)    $ (CEILING -15/4 2)
-3                  -1
```

30. If **N** is an integer, then the function **(TRUNCATE N)** returns **N**. If **N** is a positive fractional number, then the function **(TRUNCATE N)** returns the largest integer but less than **N**.

If **N** is a negative fractional number, then the **(TRUNCATE N)** function returns the smallest integer greater than **N**. In other words, the **(TRUNCATE N)** function "truncates" **N**, bringing it closer to zero.

If **N** and **M** are numbers, then the function **(TRUNCATE N M)** "truncates" to zero the quotient of **N** divided by **M**.

The function **(TRUNCATE N M)** is equivalent to the function **(TRUNCATE (/N M))**.

If **M** (the divisor) is zero, then a "Division by zero" error interrupt occurs. For example:

```
$ (TRUNCATE 7.3)     $ (TRUNCATE 8 3)
7                    2
$ (TRUNCATE -3.5)    $ (TRUNCATE -15/4 2)
-3                   -1
```

31. If **N** is an integer, the **(ROUND N)** function returns **N**. If **N** is a fractional number, the **(ROUND N)** function returns the integer closest in value to **N**. If **N** is located "in the middle" between two integers (**2*N** is an integer), the **(ROUND N)** function returns one of them that is divisible by the other without a remainder.

If **N** and **M** are numbers, then the function **(ROUND N M) rounds the quotient of N** divided by **M**. For example:

```
$ (ROUND 7.3)      $ (ROUND 8 3)
7                  3
$ (ROUND -3.5)     $ (ROUND -15/4 2)
-4                 -2
```

Calling **(ROUND N M)** is equivalent to **(ROUND (/N M))**.

If **M** (the divisor) is zero, then a "Division by zero" error interrupt occurs.

32. The **QUOTE function *("quote", "quote marks")*** prohibits the calculation of the value of its argument and returns the argument itself.

For example, we may not be interested in the value of the function call **(+ 1 2)**, which is 3, but we want to treat (+ 1 2) as a list:

```
$ (QUOTE (+ 1 2))
 (+ 1 2)
```

Let's give a couple more examples:

```
$ (QUOTE X)
X
$ (QUOTE (QUOTE 23))
```

```
   (QUOTE 23)
```

The QUOTE function does not need to be applied to constants **T, NIL** and integers, since they represent themselves!

It is important to know that:

- **the scope of the "quote mechanism" extends to the S** -expression that follows it. The "quote" **QUOTE** preceding a list prevents any attempt to evaluate the list as a function call;
- there is no mechanism to remove the "quote mark". This means that there is no way to enable the evaluation process anywhere inside the list marked with the "quote mark";
- in **muLISP-83** and later versions it is allowed to use ***the reference indicator*** **'(apostrophe) instead of the QUOTE** atom.

For example:

```
   $ '(PLUS 1 2)) $ (CAR '(CONS 4 7))
   (PLUS 1 2) CONS
```

The code for the QUOTE function in **muLISP-85** is:

```
   (DEFUN QUOTE (NLAMBDA (OBJ)
      OBJ
   ))
```

Note one more important feature of the **muLISP** interpreter: each ***new literal atom*** has a default value that matches the "printed" name of the atom.

In the next step we will introduce the concept ***of recognizers***.

Step 6.
The simplest recognizers

In this step we will introduce the concept of ***recognition*** functions.

Recognizer functions are functions used to recognize or identify **muLISP** data objects. Clearly, these functions are ***predicates***.

A predicate is a function that returns a logical value **T** or **NIL** as its result.

Let us list the simplest recognizers of the **muLISP-81** interpreter. Note that almost all predicate names end in **P (Predicate)**.

Table 1. **The simplest muLISP-81 recognizers**

Function	**Purpose**
NULL	Checking if an argument is an empty list.
ATOM	Check if an argument is an atom.
NUMBERP	Check if an argument is a numeric atom.
PLUSP	Checks if the argument is positive.
MINUS	Checks if the argument is negative.
ZEROP	Checks if the argument is null.
NOT	Logical function of negation.

AND	Logical function of multiplication.
OR	Logical addition function.

1. The **NULL** function tests whether the argument is ***an empty list***. Note that if the expression **X** is a logical value, then the **(NULL X)** function returns the logical value that is the opposite of the value of **X**. For example, the function

```
(NULL (ATOM X))
```

returns the value **T** if **X** is not an atom, and **NIL** otherwise.

2. The **ATOM** function. The syntax of this function is as follows:

```
(ATOM S-expression)
```

The ATOM predicate returns **T** if the argument is an atom and **NIL** otherwise. For example:

```
$ (ATOM 5) $ (ATOM (1 2 3))
T               NIL
$ (ATOM NIL)
T
```

3. The **NUMBERP** function. The **NUMBERP** predicate tests whether its argument is ***a numeric atom***.

4. Function **PLUSP**. Predicate **PLUSP** checks the positivity of the argument.

5. The **MINUSP** function. The **MINUSP** predicate checks whether the argument is negative.

6. The **ZEROP** function. The **ZEROP** predicate checks for the zero value of the argument.

In this section we will also note the logical functions, which are very often used in constructing complex conditional expressions. The functions **AND**, **OR** and **NOT** differ from other predicates in that their arguments, as well as their values, are **T** or **NIL**.

The number of arguments in these functions can be arbitrary.

7. If **OBJECT** is a **NIL** atom, then the function **(NOT OBJECT)** returns **T**, otherwise NIL.

Since the **NIL** atom is both an empty list and a false value, **the NULL** and **NOT** functions always return the same values. However, for better program readability, it is better to use **NULL** when checking for an empty list, and **NOT** when checking for false values. For example:

```
$ (NOT NIL)     $ (NOT 'FOO)
T               NIL
$ (NOT '())     $ (NOT (EQ 'DOG 'CAT))
T               T
```

The function code is very simple:

```
(DEFUN NOT (OBJ)
   (EQ OBJ NIL)
```

```
  )
```

8. The **(AND FORM1 FORM2 ... FORMN)** function evaluates each expression **FORM1, FORM2, ..., FORMN** in turn until one of them evaluates to **NIL** or until all expressions have been evaluated. If an expression evaluates to **NIL**, the **AND** function returns **NIL**. Otherwise, the value of the expression **FORMN** is returned.

This process is equivalent to calling the following function:

```
   (IF FORM1 (IF FORM2 (IF... (IF FORMN-1 FORMN))))
```

For example:

```
   $ (AND (EQ 'DOG 'CAT) (< 2 3))
   NIL
   $ (AND (EQ 'DOG 'DOG) (< 2 3))
   T
   $ (AND (ATOM 'DOG) (MEMBER 'DOG '(CAT DOG COW)))
   (DOG COW)
```

Note that each subsequent expression is evaluated if and only if all preceding expressions are not equal to **NIL**.

Note that the call **(AND)** returns **T**.

9. The **(OR FORM1 FORM2 ... FORMN)** function evaluates each expression in turn until one of them evaluates to non- **NIL** or until all expressions have been evaluated. If an expression evaluates to non- **NIL**, the **OR** function returns the non- **NIL** value of the expression; otherwise, **NIL** is returned.

This process is equivalent to the following function:

```
   (COND (FORM1) (FORM2)... (FORMN))
```

For example:

```
   $ (OR (EQ 'DOG 'CAT) (< 2 3))
   T
   $ (OR (EQ 'DOG 'CAT) (< 3 2))
   NIL
   $ (OR (EQ 'DOG 'DOG) (< 2 3))
   T
```

The (OR) call returns **NIL**.

Note that each subsequent expression is evaluated if and only if all expressions preceding it have evaluated to **NIL**.

In the **muLISP-85** version there are several more recognition functions: **SYMBOLP, INTEGERP, CONSP, LISTP, ODDP**. Let's talk about them.

Table 2. **The simplest muLISP-85 recognizers**

Function	**Purpose**
SYMBOLP	Check if an argument is an atom.
INTEGERP	Check if an argument is an integer.
NUMBERP	Check if an argument is a number.
CONSP	Checks whether an argument can be constructed from dotted pairs.

LISTP	Checks whether the argument is a list or an atom.
ODDP	Checks if the argument is an odd integer.

1. If **OBJECT** is ***an atom***, then the function **(SYMBOLP OBJECT)** returns **T**, otherwise **NIL**. For example:

```
$ (SYMBOL 'DOG)        $ (SYMBOL 4.25)
T                      NIL
$ (SYMBOL 100)         $ (SYMBOL '(A B C))
NIL NIL
$ (SYMBOL 2/3)
NIL
```

2. If **OBJECT** is ***an integer***, the function **(INTEGERP OBJECT)** returns **T**, otherwise **NIL**. For example:

```
$ (INTEGERP 'DOG)      $ (INTEGERP 4.23)
NIL NIL
$ (INTEGERP 1000)      $ (INTEGERP '(A D C))
T                      NIL
$ (INTEGERP 2/3)
NIL
```

3. If **OBJECT** is ***a number (integer or fractional)***, then the function **(NUMBERP OBJECT)** returns **T**, otherwise **NIL**. For example:

```
$ (NUMBER 'DOG)        $ (NUMBER 4.235)
NIL                    T
$ (NUMBER  100)        $ (NUMBER '(A B C))
T                      NIL
$ (NUMBER  2/3)
T
```

4. If **OBJECT** can be constructed from dotted pairs, then the function **(CONSP OBJECT)** returns **T**, otherwise **NIL**. For example:

```
$ (CONSP 'DOG)         $ (CONSP 4.253)
NIL NIL
$ (CONSP 100)          $ (CONSP '(A B C))
NIL                    T
$ (CONSP 2/3)          $ (CONSP NIL)
NIL NIL
```

5. If **OBJECT** is ***a list or a*** **NIL** atom, then the function **(LISTP OBJECT)** returns **T**, otherwise **NIL**. For example:

```
$ (LISTP 'DOG)         $ (LISTP 4.253)
NIL NIL
$ (LISTP 100)          $ (LISTP '(A B C))
NIL                    T
$ (LISTP 2/3)          $ (LISTP NIL)
NIL                    T
```

6. If **OBJECT** is ***an odd integer***, then the function **(ODDP OBJECT)** returns **T**, otherwise **NIL**. For example:

```
$ (ODDP 12) $ (ODDP 3.1315)
NIL NIL
$ (ODDP 0) $ (ODDP -7/3)
NIL NIL
$ (ODDP -41) $ (ODDP 'DOG)
```

```
    T                    NIL
```

In the next step we will introduce the concept ***of comparators***.

Step 7.
The simplest comparators

In this step we will introduce the concept of ***comparator*** functions.

Comparator functions are functions used to compare **muLISP** data objects. All of them require two or more arguments and, except for the **MEMBER** function, return a value of **T** or **NIL**, i.e., they are ***predicates***.

Here is a list of these functions.

Table 1. **Simplest comparators muLISP-81**

Function	**Purpose**
EQ	Checks if objects are in the same memory area.
EQUAL	Checks the identity of two S-expressions.
MEMBER	Checks if the given object is in the list.
LESSP	Checks which of the numeric arguments is smaller.
GREATERP	Checks which of the numeric arguments is greater.
ORDERP	Checks the order of arguments.

1. Function **EQ**. The predicate **EQ** determines whether two **LISP** objects in computer memory are physically represented by the same structure, i.e. whether they are located in the same memory location. For example:

```
    $ (EQ 5 5)      $ (EQ A A)
    T               T
```

Since **EQ** is defined only for ***atoms***, to compare two expressions, you must first determine whether they are atoms. If at least one of the arguments is a list, the **EQ** predicate cannot be used for logical comparison. For example:

```
    $ (EQ (1 2 3) (1 2 3))
    NIL
```

2. The **EQUAL** function. The **EQUAL** predicate checks the identity of two S-expressions (their ***logical*** equivalence). For example:

```
    $ (EQUAL (1 2 3) (1 2 3))
    T
```

The EQUAL predicate is a generalization of the **EQ** function: it applies to arbitrary list structures. Use it when you are unsure of the type of objects being compared or the correctness of using the **EQ** predicate.

3. MEMBER function. Syntax functions like this:

```
              (MEMBER OBJECT LIST)
```

The MEMBER predicate checks whether the given **OBJECT** is in **the LIST**. It is not enough that the first argument is "buried" somewhere in the second argument: it must be a top-level element of the second argument. For example:

```
    $ (MEMBER A ((A B) (C D)))
    NIL
```

Note that the functions **EQ, EQUAL, MEMBER** are sometimes called ***primitive pattern matching functions***.

4. LESSP function. The **LESSP** predicate checks for numeric arguments which of them is smaller.

5. GREATERP function. The **GREATERP** predicate checks for numeric arguments which of them is greater.

6. ORDERP function. The **ORDERP** predicate has two arguments (both of which can be either atoms, numbers, or dotted pairs). The predicate returns **T** if the first argument "comes before" the second argument, otherwise the function returns **NIL**.

Note that the numbers are "placed before" the atoms; they in turn are placed before the dot pairs. The two numbers are placed in ascending order. The two atoms are "placed" according to the result of comparing their addresses in physical memory (i.e., according to the order in which they were entered). The two dot pairs are similarly placed. For example:

```
    $ (ORDERP 5 9) $ (ORDERP DOG CAT)
    T                  T
    $ (ORDERP DOG 5) $ (ORDERP CAT DOG)
    NIL NIL
```

The ORDERP function provides an optimal "chronological" arrangement of atoms when writing sorting programs.

There are several more comparator functions in the **muLISP-85 version: EQL, MEMBER-IF, =, /=, <, >, <=, >=, MISMATCH**. Let's look at them in more detail.

Table 2. **The simplest muLISP-85 recognizers**

Function	**Purpose**
EQ	Checking the identity of two objects.
IQ	Checking the identity of two objects.
EQUAL	Checking the identity of two objects.
MEMBER	Find an item in a list.
(= N1 N2... Nm)	Pairwise test for equality of arguments.
(/= N1 N2... Nm)	Pairwise test for inequality of arguments.
(< N1 N2... Nm)	Pairwise test for increasing arguments.
(> N1 N2... Nm)	Pairwise test for decreasing arguments.
(<= N1 N2... Nm)	Pairwise test for non-increasing arguments.
(>= N1 N2... Nm)	Pairwise test for non-decreasing arguments.
MISMATCH	Compare list items.

1. If **OBJECT1** is identical to **OBJECT2**, then the function **(EQ OBJECT1 OBJECT2)** returns **T**, otherwise **NIL**.

Function **(NEQ OBJECT1 OBJECT2)** is equivalent to functions **(NOT (EQ OBJECT1 OBJECT2))**.

The EQ function tests whether its two arguments are identical (point to the same data objects in memory). Since characters and ***small integers*** (i.e. integers less than 65536 in absolute value) are uniquely defined, the **EQ** function is a good way to determine whether two characters or two small integers are equal to each other. For example:

```
$ (EQ 'Alice 'Alice)    $ (EQ FOO BAR)
T                       T
$ (SETQ FOO '(A B C))   $ (EQ 7 (+ 3 4))
(A B C)                 T
$ (EQ FOO '(A B C))     $ (EQ 100000 100000)
NIL NIL
$ (SETQ BAR FOO)
(A B C)
```

Here is the function code:

```
(DEFUN EQ (OBJ1 OBJ2)
   (= (LOCATION OBJ1) (LOCATION OBJ2))
)
```

2. If **OBJECT1** is identical to **OBJECT2** or if both arguments are numbers with the same values, then the function **(EQL OBJECT1 OBJECT2)** returns **T**, otherwise **NIL**.

Function **(NEQL OBJECT1 OBJECT2)** equivalent functions **(NOT (EQL OBJECT1 OBJECT2))**.

Because ***large integers*** (integers greater than 65536 in absolute value) and fractional numbers are not uniquely determined, the **EQL** function can be much more efficient than the **EQ** function at determining whether two numbers are equivalent. Non-atomic arguments are equivalent if and only if they point to the same dot pairs in memory. For example:

```
$ (EQL 'Alice 'Alice)    $ (EQL 100000 100000)
T                        T
$ (EQL 7 (+ 3 4))        $ (EQL '(A B) '(A B))
T                        NIL
```

Here is the function code:

```
(DEFUN EQL (OBJ1 OBJ2)
   ( (AND (NUMBERP OBJ1) (NUMBERP OBJ2))
       (=OBJ1 OBJ2)
   (= (LOCATION OBJ1) (LOCATION OBJ2))
)
```

3. If **OBJECT1** is equivalent to **OBJECT2**, then the function **(EQUAL OBJECT1 OBJECT2)** returns **T**, otherwise **NIL**. The **EQUAL** function checks its two arguments for equivalence.

If one or both arguments are ***atoms***, the **EQUAL** function behaves like the **EQL** function.

Two non-atom arguments are in an **EQUAL** relation if they are isomorphic data structures: both **the CAR** branch and **the CDR** branch are **EQUAL**. Two **EQUAL** data structures are printed identically. Because **EQL** is faster, it is better to use this function instead of **EQUAL** where at least one of the arguments is known to be an atom. For example:

```
$ (EQUAL 'A 'A)
T
$ (EQUAL '(A B C) '(A B C))
```

```
    T
    $ (EQUAL '(A B C) '(C B A))
    NIL
```

Here is the function code:

```
    (DEFUN EQUAL (OBJ1 OBJ2)
       ( (ATOM OBJ1) (EQL OBJ1 OBJ2) )
       ( (ATOM OBJ2) NIL )
       ( (EQUAL (CAR OBJ1) (CDR OBJ2))
            (EQUAL (CDR OBJ1) (CDR OBJ2)) )
    )
```

4. The function **(MEMBER OBJECT LIST TEST)** performs a linear search in **LIST** for an element for which the comparison flag with **OBJECT** by the **TEST** test is not equal to **NIL**. If **the TEST** argument is **NIL** or is not specified, the **MEMBER** function uses **the EQL** test.

The (MEMBER-IF TEST LIST) function searches the **LIST** list for an element for which the test flag for the **TEST** test is not **NIL**. For both functions, if an element that satisfies the **TEST** test is found, the tail of the **LIST** list, starting with that element, is returned. Otherwise, **NIL** is returned. For example:

```
    $ (MEMBER 'A '(B C D))
    NIL
    $ (MEMBER 'A '(B A D))
    (A D)
    $ (MEMBER-IF 'NUMBERP '(A B 3 C (-7)))
    (3 C (-7))
    $ (MEMBER-IF 'MINUSP '(A B 3 C (-7)))
    NIL
```

The function code is as follows:

```
    (DEFUN MEMBER (OBJ LST TEST)
       ( (ATOM LST) NIL)
       ( ( (NULL TEST) (SETQ TEST 'EQL) )
         ( (FUNCALL TEST OBJ (CAR LST)) LST )
       (MEMBER OBJ (CDR LST) TEST)
    )
```

5. If **N1** is equal to **N2**, **N2** is equal to **N3**, ..., **Nm-1** is equal to **Nm**, then the function **(= N1 N2... Nm)** returns **T**, otherwise **NIL**. The function raises the "Non-numeric argument" interrupt if any argument is not a number. For example:

```
    $ (= 5 9)        $ (= 3 3.0)
    NIL              T
    $ (= 4 4 -7)     $ (= 0.75 6/8)
    NIL              T
```

6. If **N1** is not equal to **N2**, **N2** is not equal to **N3**, ..., **Nm-1** is not equal to **Nm**, then the function **(/= N1 N2... Nm)** returns **T**, otherwise **NIL**. For example:

```
    $ (/= 5 9)       $ (/= 3 3.0)
    T                NIL
    $ (/= 4 4 -7)    $ (/= 0.75 6/8)
    NIL NIL
```

Note that if the function returns **T**, it means that there is no pair of adjacent equal arguments. However, this can also be the case when two arguments are equal but not adjacent. For example:

```
    $ (/= 6 -2 6)
    T
```

If any argument is not a number, a “Non-numeric argument” error occurs.

Let's give the implementation of the function:

```
(DEFUN /= LST
   ( (NULL LST) )
   ( (NULL (CDR LST)) )
   ( (AND (NUMBER (CAR LST)) (NUMBER (CADR LST)))
        ((= (CAR LST) (CADR LST)) NIL)
        (APPLY '/= (CDR LST)) )
   ((BREAK (LIST '/= (CAR LST) (CADR LST))
           'Nonnumeric Argument') NIL)
   (APPLY '/= (CDR LST))
)
```

7. If **N1<N2, N2<N3, ..., Nm-1<Nm**, then the function **(< N1 N2... Nm)** returns **T**, otherwise **NIL**. For example:

```
$ (< 5 9)      $ (< 3 3.0)
T              NIL
$ (< 4 -7)     $ (< 3/5 2/3)
NIL            T
```

If any argument is not a number, a “Non-numeric argument” error occurs.

The function is useful when you need to determine whether a number is within a given range; it is called with three arguments: the number and the range boundaries. For example, here's how to determine whether a value is **CHAR**:

```
by number: (< 47 (ASCII CHAR) 58)
uppercase: (< 64 (ASCII CHAR) 91)
lowercase letter: (< 96 (ASCII CHAR) 123)
```

8. If **N1>N2, N2>N3, ..., Nm-1>Nm**, then the function **(> N1 N2... Nm)** returns **T**, otherwise **NIL**. For example:

For example:

```
$ (> 5 9)      $ (> 3 3.0)
NIL NIL
$ (> 4 -7)     $ (> 3/5 2/3)
T              NIL
```

If any argument is not a number, a "Non-numeric argument" error interrupt occurs.

9. If **N1<=N2, N2<=N3, ..., Nm-1<=Nm**, then the function **(<= N1 N2... Nm)** returns **T**, otherwise **NIL**. For example:

```
$ (<= 5 9)     $ (<= 3 3.0)
T              T
$ (<= 4 -7)    $ (<= 3/5 2/3)
NIL            T
```

If any argument is not a number, a “Non-numeric argument” error occurs.

Here is the function code:

```
(DEFUN <= LST
      ((NULL LST))
      ((NULL (CDR LST)))
      ((AND (NUMBERP (CAR LST)) (NUMBERP (CADR LST)))
          ((> (CAR LST) (CADR LST)) NIL)
          (APPLY '<= (CDR LST)) )
      ((BREAK (LIST '<= (CAR LST) (CADR LST))
           "Nonnumeric Argument")
          (APPLY '<= (CDR LST)) )
   )
```

10. If **N1>=N2, N2>=N3, ..., Nm-1>=Nm**, then the function **(>= N1 N2... Nm)** returns **T**, otherwise **NIL**. For example:

```
   $ (>= 5 9)       $ (>= 3 3.0)
   NIL              T
   $ (>= 4 -7)      $ (>= 3/5 2/3)
   T                NIL
```

This function generates a "Non-numeric argument" error trap if any argument is not a number.

11. The **(MISMATCH LIST1 LIST2 TEST)** function compares the elements of **LIST1** and **LIST2** using the two-element predicate **TEST** until a mismatch is encountered (i.e., until the predicate returns **NIL**) or until the end of one or both lists is reached. If **TEST** is **NIL** or not specified, the **MISMATCH** function uses **the EQL** test.

If the lists are of equal length and no mismatch is found, the function returns **NIL**. If the lists are of different lengths and no mismatch is found, the function returns the length of the shorter list. Otherwise, the function returns the ordinal number, starting with zero, of the first pair of list elements that do not match. For example:

```
   $ (MISMATCH '(A B C D E) '(A B C D E))
   2
   $ (MISMATCH '(5 -4 6 2 ) '(7 -3 9 8) '<)
   NIL
```

In the next step, we will begin to explore the functions used to work with lists.

Step 8.
Basic functions for working with lists. Elementary selectors

In this step, we will begin to explore the functions used to work with lists.

In the **LISP** language, there are basic functions for constructing and analyzing lists. They are used to describe the system of axioms of the language (the algebra of list processing).

The basic functions are: **CAR, CDR, CONS, ATOM** and **EQ**. Note that we have already considered the predicates **ATOM** and **EQ**.

Table 1. **Functions for working with lists in muLISP-81**

Function	**Purpose**
CAR	Returns the first element of the list (the head of the list).
CDR	Returns a list without the first element (the tail of the list).
MOLD	Returns the head of the list head.
CDAR	Returns the tail of the head of the list.

CADR	Returns the head of the tail (the second element of the list).
CDDR	Returns the tail of the list.
CADDR	Returns the head of the tail of the tail (the third element of the list).
CDDDR	Returns the tail of the tail of the list.
AAR	Returns the head of the list head's head.
CDAR	Returns the tail of the head of the list.
PERFUME	Returns the head of the tail of the list head.
CDDAR	Returns the tail of the list head.
CAADR	Returns the head of the tail of the list.
CDADR	Returns the tail of the head of the tail of the list.

1. The **CAR function (pronounced "kar") returns the first element of a list (the head of the list)** as its value. For example:

```
    $ (CAR (A B C D E))
    A
    $ (CAR (A.B) . C))
    (A.B)
```

If a literal atom is used as an argument to the **CAR** function, the result of the function execution depends on the specific dialect. In **muLISP,** applying **CAR** to an atom is allowed, in other dialects (e.g., **Common LISP, Standard LISP**) this will cause an error message.

2. The **CDR** function (if the name has no vowels, like in **CDR**, then we do our best to make it easier to pronounce, the word **CDR** can be pronounced something like "cudder" or "kud-er"), applied to ***a list***, returns a list obtained from the original by discarding the first element (***the tail of the list***), for example:

```
    $ (CDR (A B C D E))
    (B C D E)
    $ (CDR ((A.B) . C))
    C
```

The tail of a list with one element is ***the empty list*** **(NIL)**.

Note that in **muLISP-85 the CDR** element of a number is ***the attribute and type of the number***.

The CDR function of ***an atom*** returns ***a list of the properties*** of that atom.

The unusual names of the **CAR** and **CDR** functions arose for historical reasons. These names are abbreviations of the **Contents of Address portion of Register (CAR)** and **Contents of Decrement portion of Register (CDR) registers of the IBM 704** computer. The author of **the LISP** language, John McCarthy (USA), implemented the first **LISP** system on this machine and used the **CAR** and **CDR** registers to store pointers to the head and tail of the list.

Note that by combining the **CAR** and **CDR** selectors, you can select list items. For example:

```
    $ (CDR (CDR (CAR ((A B C) (D E) (F)))))
    (C)
```

Combinations of **CAR** and **CDR** calls form deep list calls, and **muLISP** uses a shorter notation for this: the desired combination of **CAR** and **CDR** calls can be written as a single function call: **(C...R List)**.

Instead of the ellipsis, the required combination of letters **A** (for the **CAR** function) and **D** (for the **CDR** function) is written, for example: instead of **(CADR X), (CAR (CDR X))** is written.

3. **CAAR** - head of the list head.

4. **CDAR** - tail of the list head.

5. **CADR** - head of tail (***second element of the list***).

6. **CDDR** - tail of the tail of the list.

7. **CADDR** - head tail tail (***third element of the list***).

8. **CDDDR** - tail of tail of tail of list.

9. **CAAAR** - head of the head of the head of the list.

10. **CDAAR** - list head tail.

11. **CADAR** - head tail head list.

12. **CDDAR** - tail of the tail of the head of the list.

13. **CAADR** - head of the head of the tail of the list.

14. **CDADR** - tail of the head of the tail of the list.

Note that in the **muLISP-81** dialect it is no longer possible to write the expression for ***the fourth element of the list*** in abbreviated form: you have to write **(CAR (CDDDR LST))**.

In the next step we will continue our discussion of selectors, in particular we will list the selectors available in **muLISP-85**.

Step 9.
Basic functions for working with lists. Elementary selectors (continued)

In this step we will continue our discussion of selectors, in particular we will provide a list of selectors available in **muLISP-85**.

In the **muLISP-85** and **muLISP-87** dialects, there are about ten more selector functions defined: **LAST, NTHCDR, NTH, SUBLIST, COUNT, COUNT-IF, FIND, POSITION**. It is interesting to note that **the ASSOC, RASSOC** functions are also selectors! Let's talk about selector functions in **muLISP-85**.

Table 1. **Functions for working with lists in muLISP-85**

Function	**Purpose**
(LAST LIST)	Returns the last top-level dotted pair of the **LIST**.
(NTHCDR N LIST)	Returns **the N-th CDR** of the list **LIST**.
(NTH N LIST)	Returns **the N-th** element of the list **LIST**.
(COUNT OBJECT LIST TEST)	Returns the number of elements in the **LIST** list for which the attribute, when compared to the **OBJECT** object by the **TEST** test, is not equal to **NIL**.
(SUBLIST LIST N M)	Copies and outputs **the N-th** through **M-th** elements of **LIST**, where **the CAR** element **of**

	LIST is the zeroth element.
CAAR, ..., CDDR, ..., CAAR, ..., CAAR, ..., CDDDDR	Combinations of **CAR** and **CDR**.

1. The **(LAST LIST)** function returns the last top-level dotted pair of the **LIST** list. Note that the **LAST** function returns the last ***dotted pair***, not the last element **of the LIST list**. If the **LIST list is an atom, the LAST** function returns **NIL**. For example:

```
    $ (LAST '(A B C D))       $ (LAST 'FCO)
    (D) NIL
    $ (LAST '(A B C. D))     $ (LAST '((2 1) (1 2) (3 4)))
    (C. D)                   ((3 4))
```

The last element of a list can be determined by applying the **CAR** function to the **(LAST LIST)** argument. For example:

```
    $ (CAR (LAST '(A B C)))
    C
```

Here is the function code:

```
   (DEFUN LAST (LST)
      ( (ATOM LST) NIL )
      ( (ATOM (CDR LST)) LST )
      ( LAST (CDR LST) )
   )
```

2. If **N** is zero or a positive integer, then the **(NTHCDR N LIST)** function returns **the N-th CDR** of **LIST**. The **NTHCDR** function returns **NIL** if **N** is neither zero nor a positive integer, or if **LIST** has **N** or fewer elements. For example:

```
    $ (NTHCDR 0 '(A B C D))      $ (NTHCDR 5 '(A B C D))
    (A B C D)                    NIL
    $ (NTHCDR 1 '(A B C D))      $ (NTHCDR 2 '(A B. C))
    (B C D)                      C
    $ (NTHCDR 2 '(A B C D))
    (C D)
```

Here is the function code:

```
   (DEFUN NTHCDR (N LST)
      ( (ZEROP N) LST )
      ( (AND (INTEGERP N) (PLUSP N))
           ( (ATOM LST) NIL )
           ( NTHCDR (SUB1 N) (CDR LST)) )
   )
```

3. If **N** is zero or a positive integer, then **(NTH N LIST)** returns **the N-th** element of **LIST**, where **CAR** of **LIST** is considered to be the zeroth element. The **NTH** function returns **NIL** if **N** is neither zero nor a positive integer, or if **LIST** has **N** or fewer elements. For example:

```
    $ (NTH 0 '(A B C D))         $ (NTH 4 '(A B C D))
    A                            NIL
    $ (NTH 3 '(A B C D))         $ (NTH 2 '(A B. C))
    D                            NIL
```

Here is the function code:

```
   (DEFUN NTH (N LST)
      ( (ATOM (NTHCDR N LST)) NIL )
      ( CAR (NTHCDR N LST) )
   )
```

4. The function **(COUNT OBJECT LIST TEST)** returns the number of elements in the list **LIST** for which the attribute is not equal to **NIL when compared with the object OBJECT** by the test **TEST**. If the test argument is not specified or is equal to **NIL**, the **COUNT** function uses **the EQL** test.

The (COUNT-IF TEST LIST) function returns the number of elements in the **LIST** list for which the test flag for the **TEST** test is not **NIL**. For example:

```
   $ (COUNT 'DOG '(CAT DOG COW PIG DOG ANT))
   2
   $ (COUNT-IF 'EVENP '(3 -6 8 7 0))
   3
   $ (COUNT 'a '(NIL a (2 a) a a (3 4)))
   3
   $ (COUNT-IF 'NUMBERP '(NIL 3 (2 a) a -14 (3 4)))
   2
```

5. If **N** and **M** are non-negative integers, and **N<=M**, then the function **(SUBLIST LIST N M)** copies and returns the **N-th** through **M-th** elements of the list **LIST**, where **the CAR** element **of LIST** is the zero element.

If **M** is not an integer or is greater than or equal to the length of LIST, M **is** taken to be one less than the length **of LIST**. If **N** is not an integer, a negative number, or **N>M**, **SUBLIST** returns **NIL**. For example:

```
   $ (SUBLIST '(A B C D E F) 2 4)
   (C D E)
   $ (SUBLIST '(A B C D E F) 2 2)
   (C)
   $ (SUBLIST '(A B C D E F) 0 3)
   (A B C D)
   $ (SUBLIST '(A B C D E F) 2)
   (C D E F)
```

Here is the function code:

```
   (DEFUN SUBLIST (LST N M)
      ( (INTEGERP N)
         ( (INTEGERP M)
             (FIRST (ADD1 (-M N)) (NTHCDR N LST)) )
         (NTHCDR N LST) )
   )
```

6. **Functions that are combinations of CAR** and **CDR** are allowed. The name of each of these functions begins with "C" and ends with "R". Between them is a sequence of letters "A" and "D", indicating how **CAR** and **CDR** are combined when executing the function. The following combinations are allowed in **muLISP:**

```
   CAAR, ..., CDDR, ..., CAAR, ..., CAAR, ..., CDDDDR.
```

Let us consider typical definitions for composite selection functions:

```
   (DEAD CAR (OBJ)
      (CAR (CAR OBJ))
   )
   (DEFUN CADDR (OBJ)
      (CAR (CDR (CDR OBJ)))
   )
   (DEFUN CDADAR (OBJ)
      (CDR (CAR (CDR (CAR OBJ))))
   )
```

The following selectors are widely used to extract list items:

```
   $ (CAR '(A B C D E))              $ (CADDR '(A B C D E))
   A                                 C
   $ (CDR '(A B C D E))              $ (CADDDR '(A B C D E))
   (B C D E)                     D
   $ (CADR '(A B C D E))
   B
```

Note that these selectors can be given other mnemonic names (if desired). For this, you can use the **MOVD** function, which will replace the definition of **CAR** with **FIRST**, **CDR** with **REST**, **CADR** with **SECOND**, etc.

In the next step we will study ***elementary constructors***.

Step 10.
Basic constructors for working with lists

This step is dedicated to getting acquainted with elementary constructors.

Table 1. **Elementary constructors**

Function	**Purpose**
CONS	Builds a list (dotted pair) from two arguments.
LIST	Returns a list of the argument values.
REVERSE	Creates an "inverted" list.
OBLIST	Creates and returns a list of objects, i.e. a list of currently active symbols in the system.
(LENGTH OBJECT)	Returns the "length" **of OBJECT** corresponding to its type.

1. The **CONS** function allows you to build a list (dotted pair) from two arguments: the first argument is an S-expression, the second argument is a list. The syntax for calling the function is:

```
      (CONS S-expression List)
```

The CAR and **CDR** selectors are the inverse of the **CONS** constructor. A list "split" into a head and tail by the **CAR** and **CDR** functions can be reconstructed using the **CONS** function. For example:

```
   $ (CONS (CAR (1 2 3)) (CDR (1 2 3)))
   (1 2 3)
```

The second argument can also be an atom, but in this case the function returns ***a dotted pair***. For example:

```
   $ (CONS A B)
   (A. B)
```

By definition of the **CONS** function, a list consisting of one element is formed as follows:

```
   $ (CONS A NIL)
   (A)
```

With this in mind, list construction can easily be reduced to nested calls to the **CONS** function, with the second argument of the last call being **NIL**, which serves as the basis for list growth. For example:

```
   $ (CONS A (CONS B (CONS C NIL)))
   (A B C)
```

2. The **LIST** function returns a list of the argument values. The number of arguments is arbitrary. This creates a new list nested one level deeper in parentheses than any of its arguments. For example:

```
    $ (LIST 1 2 3)
    (1 2 3)
    $ (LIST (1 2 3) 4 (5))
    ((1 2 3) 4 (5))
```

3. The **REVERSE** function allows you to create an "inverted" list. Note that the original list is not changed! For example:

```
    $ (REVERSE (1 2 3))
    (3 2 1)
```

4. The **OBLIST** function creates and returns a list of objects, i.e. a list of currently active symbols in the system. The symbols are arranged in the order in which they were read and/or generated: newer symbols are arranged ***"to the left"*** of older ones. Note that unnecessary symbols are automatically removed ***by the garbage collector***.

The function call looks like this:

```
    $ (OBLIST)
```

The object list is used when a new object appears to determine whether the object is really new or has already been encountered in the program.

5. The **(LENGTH OBJECT)** function returns the "length" **of an OBJECT** corresponding to its type.

If **OBJECT** is **NIL** or a dotted pair, the **LENGTH** function returns the number of high-level dotted pairs (i.e., elements) in the object.

If **OBJECT is an atom, the LENGTH** function returns the number of characters in the P-name of **OBJECT**.

If **OBJECT** is a number, the **LENGTH** function returns the number of words (a word is 2 bytes) required to hold the numeric vector **OBJECT**. For example:

```
    $ (LENGTH (A B C D))         $ (LENGTH MULISP)
    4                            6
    $ (LENGTH NIL)               $ (LENGTH -13)
    0                            3
```

In the next step we will continue to study constructors.

Step 11.
Basic constructors for working with lists (continued)

In this step we will continue to study elementary constructors.

In muLISP-85 and **muLISP-87** dialects, about 20 ***constructor*** functions are defined: **ACONS, LIST*, APPEND, COPY-LIST, COPY-TREE, FIRSTN, BUTLAST, REMOVE, REMOVE-IF, SUBSTITUTE, SUBSTITUTE-IF, SUBST, SUBST-IF, MAKE -LIST** etc.

Table 1. **Elementary constructors in muLISP-85**

Function	Purpose
(CONS OBJECT1 OBJECT2)	Returns a dotted pair whose **CAR** element points to **OBJECT1** and whose **CDR** element points to **OBJECT2.**
(LIST OBJECT1... OBJECTN)	Creates and returns a list consisting of elements **OBJECT1** through **OBJECTN**.
(LIST* OBJECT1... OBJECTN)	Binds objects **OBJECT1, ...,OBJECTN** into a pair and returns the resulting object.
(APPEND LIST1 LIST2... LISTN)	Creates and returns a list consisting of the elements of lists starting with list **LIST1** and going through list **LISTN**.
(COPY-LIST LIST)	Copies the top-level dotted pairs of **LIST** and returns a list that is **EQUAL** to **LIST**.
(COPY-TREE OBJECT)	Copies all dotted pairs of the **OBJECT** and returns an object equivalent to **OBJECT**.
(FIRST N LIST)	Copies and returns the first **N** elements of a list.
(BUTLAST LIST N)	Copies and returns all but the last **N** elements of **LIST**.
(REVERSE LIST)	Creates and returns a list consisting of the elements of **LIST**, but in reverse order.
(SUBSTITUTE NEW OLD LIST TEST)	Returns a high-level copy of the **LIST**.
(MAKE-LIST N OBJECT LIST)	Creates and returns a list of **N** elements.
REMOVE	Removes items from a list.
SUBST	Replaces items in a list.

1. The **(CONS OBJECT1 OBJECT2)** function returns a dotted pair whose **CAR** element points to **OBJECT1** and whose **CDR** element points to **OBJECT2**. If the value of the ***FREE-LIST*** system variable is a dotted pair, the **CONS** function modifies and returns this dotted pair and changes ***FREE-LIST*** to **(CDR *FREE-LIST*)**.

The correct interpretation of the function **(CONS OBJECT1 OBJECT2)** depends on how **OBJECT2** is viewed.

If **OBJECT2** is a list, the **CONS** function creates a list whose first element is **OBJECT1** and whose remainder is **OBJECT2**.

If **OBJECT2** is an atom or a binary tree, the **CONS** function creates a tree whose left branch (**the CAR** branch) is **OBJECT1** and whose right branch (**the CDR** branch) is **OBJECT2**. For example:

```
$ (CONS 'A '(B C D))
(A B C D)
$ (CONS 'A 'B)
(A. B)
```

2. The **(LIST OBJECT1... OBJECTN)** function creates and returns a list consisting of elements from **OBJECT1** to **OBJECTN**. If the function is called without arguments, the **LIST** function returns the **NIL** flag. The **LIST** function can work with any number of arguments. For example:

```
$ (LIST 'A 'B 'C 'D)      $ (LIST)
(A B C D)                 NIL
$ (LIST 'A '(B C) 'D)
(A (B C) D)
```

Here is the function code:

```
(DEFUN LIST LST
   ( (NULL LST) NIL )
   ( CONS (CAR LST) (APPLY 'LIST (CDR LST)) )
)
```

3. The function **(LIST* OBJECT1... OBJECTN)** pairs the objects **OBJECT1, ...,OBJECTN** and returns the resulting object. For example:

```
    $ (LIST* 'A 'B 'C 'D)
    (A B C. D)
    $ (LIST* 'A 'B '(C D))
    (A B C D)
```

If a function is called with a single argument, **LIST*** returns that argument. For example:

```
    $ (LIST* 'DOG)
    DOG
```

Here is the function code:

```
    (DEFUN LIST* LST
       ( (NULL LST) NIL )
       ( (NULL (CDR LST)) (CAR LST) )
       ( CONS (CAR LST) (APPLY 'LIST*  (CDR LST)) )
    )
```

4. The **(APPEND LIST1 LIST2... LISTN)** function creates and returns a list consisting of the elements of the lists starting with **LIST1** and ending with **LISTN**. The **APPEND** function copies the top-level dotted pairs of each of its arguments except the last.

If a function is called with a single argument, **APPEND** returns that argument without copying it. Therefore, it is better to use the **COPY-LIST** function than the **APPEND** function to copy a list.

Note that if **the APPEND** and **NCONC** functions have the same arguments, they generally return the same lists. However, the **APPEND** function uses dotted pairs to copy all but the last argument, whereas the **NCONC** function actually modifies all but the last argument. For example:

```
    $ (SETQ FOO '(D E F))
    (D E F)
    $ (APPEND '(A B C) FOO '(G H I))
    (A B C D E F G H I)
    $ FOO
    (D E F)
```

Here is the function code:

```
    (DEFUN APPEND (LST1 LST2)
       ( (ATOM LST1) LST2 )
       ( CONS (CAR LST1) (APPEND (CDR LST1) LST2) )
    )
```

5. The **(COPY-LIST LIST)** function copies the top-level dotted pairs of a LIST **and** returns a list that is EQUAL **to** the **LIST**. For example:

```
    $ (COPY-LIST '(A B (C . D) E))
    (A B (C . D) E)
    $ (COPY-LIST '(A B C . D))
    (A B C . D)
```

Here is the function code:

```
    (DEFUN COPY-LIST (LST)
       ( (ATOM LST) LST )
       ( CONS (CAR LST) (COPY-LIST (CDR LST)) )
    )
```

6. The **(COPY-TREE OBJECT)** function copies all dotted pairs of the **OBJECT** object and returns an object equivalent to **OBJECT**. For example:

```
   $ (COPY-TREE '(A B (C . D) E))
   (A B (C . D) E)
   $ (COPY-TREE '(A B C . D))
   (A B C . D)
```

Here is the function code:

```
   (DEFUN COPY-TREE (OBJ)
      ( (ATOM OBJ) OBJ )
      ( CONS (COPY-TREE (CAR OBJ)) (COPY-TREE (CDR OBJ)) )
   )
```

7. If **N** is a positive integer, then the function **(FIRST N LIST)** copies and returns the first **N** elements of the list. The **FIRSTN** function returns **NIL** if **N** is a non-positive integer.

If **LIST** has **N** or more elements, the **FIRSTN** function copies and returns the elements of **LIST**. For example:

```
   $ (FIRSTN 0 '(SUE JOE ANN BOB))
   NIL
   $ (FIRSTN 2 '(SUE JOE ANN BOB))
   (SUE JOE)
   $ (FIRSTN 4 '(SUE JOE ANN BOB))
   (SUE JOE ANN BOB)
   $ (FIRSTN 5 '(SUE JOE ANN BOB))
   (SUE JOE ANN BOB)
   $ (FIRSTN 2 '(A B . C))
   (A B)
   $ (FIRSTN 3 '(A B . C))
   (A B)
```

The function code looks like this:

```
   (DEFUN FIRSTN (N LST)
      ( (AND (INTEGERP N) (PLUSP N))
          ( (ATOM LST) NIL )
          ( CONS (CAR LST) (FIRSTN (SUB1 N) (CDR LST))) )
   )
```

8. If **N** is zero or a positive integer, then the **(BUTLAST LIST N)** function copies and returns all but the last **N** elements of **LIST**. For example:

```
   $ (BUTLAST '(A B C D) 2)
   (A B)
```

If **N** is omitted or is either zero or a non-positive integer, the **BUTLAST** function copies and returns all but the last element of **LIST**. For example:

```
   $ (BUTLAST '(A B C D))
   (A B C)
```

Here is the function code:

```
   (DEFUN BUTLAST (LST N)
      ( (AND (INTEGERP N) (>=N 0))
          (FIRST (-(LENGTH LST) N) LST) )
      (BUTLAST LST 1)
   )
```

9. The **(REVERSE LIST)** function creates and returns a list consisting of the elements of the **LIST** list, but in reverse order.

The (REVERSE LIST OBJECT) function returns the elements of **the LIST** list in reverse order, appended to **OBJECT**. For example:

```
    $ (REVERSE '(A B C D E))                 $ (REVERSE '(A B C) 'D)
    (E D C B A)                              (C B A . D)
    $ (REVERSE '(A B C) '(D E F))
    (C B A D E F)
```

The result is the same as running the **(APPEND (REVERSE LIST) OBJECT)** function, but calling **REVERSE** as the second argument is more efficient.

Here is the function code:

```
    (DEFUN REVERSE (LST OBJ)
       ( (ATOM LST) OBJ )
       ( REVERSE (CDR LST) (CONS (CAR LST) OBJ)))
    )
```

10. The function **(SUBSTITUTE NEW OLD LIST TEST)** returns a high-level copy of the list **LIST**, replacing with **NEW** elements those **OLD** elements of the list **LIST** for which the verification flag for the **TEST** test is not **NIL**. If the test argument is **NIL** or is not specified, **SUBSTITUTE** uses **the EQL** test.

The (SUBSTITUTE-IF NEW TEST LIST) function returns a high-level copy of the **LIST** list, replacing all elements of the **LIST list for which the TEST** test flag is not **NIL with NEW** elements. For example:

```
    $ (SUBSTITUTE 5 2 '(4 2 (3. 2) 4))
    (4 5 (3 . 2) 4)
    $ (SUBSTITUTE 'CANNIBALS 'NOUN '(NOUN LIKE TO EAT NOUN))
    (CANNIBALS LIKE TO EAT CANNIBALS)
```

11. The function **(MAKE-LIST N OBJECT LIST)** creates and returns a list of **N** elements, each of which takes the value of **OBJECT**, appended to the list **LIST**. The absence of **N** is identified with 0, the absence of **OBJECT** and **LIST** is identified with **NIL**. For example:

```
    $ (MAKE-LIST 4)                       $ (MAKE-LIST 3 5 '(2 3))
    (NIL NIL NIL NIL)                     (5 5 5 2 3)
    $ (MAKE-LIST 3 '(A B C))
    ((A B C) (A B C) (A B C))
```

Here is the function code:

```
    (DEFUN MAKE-LIST (N OBJ LST)
       ( (AND (INTEGERP N) (PLUS N))
            (CONS OBJ (MAKE-LIST (SUB1 N) OBJ LST)) )
           LST )
```

12. Let us now describe the constructors **REMOVE** and **SUBST**. The first function is used to remove elements from the list, and the second one is used to replace elements. The functions **REMOVE** and **SUBST** are implemented in a conditional version. This means that one more argument is added, which is a predicate, the truth of which is checked before applying the function to the elements of the list being modified. If the result of the check is true, then the function is executed, otherwise it is not.

The (REMOVE ELEMENT LIST TEST) function returns a copy of the **LIST** list with all elements except those that, when tested by the **TEST** test, have a non- **NIL** attribute and are removed (**(TEST ELEMENT)** is not **NIL**).

If **TEST** is **NIL** or not specified, the **REMOVE** function uses **the EQL** test.

The (REMOVE-IF TEST LIST) function returns a copy of the **LIST list with all elements except those that have a non- NIL** test value (**(TEST ELEMENT)** does not return **NIL**) and are removed. For example:

```
    $ (REMOVE '(2 5) '((5 2) (2 5) (2 3)) 'EQUAL)
    ((5 2) (2 3))
    $ (REMOVE-IF 'MINUSP '(-2 0 7 -0.1 3))
    (0 7 3)
```

The (SUBST NEW OLD OBJECT TEST) function returns a copy of the **OBJECT** object, replacing with **NEW** all **OLD** subexpressions of the **OBJECT** object for which the test flag of **the TEST** test is not **NIL**. If **TEST** is **NIL** or is not specified, **SUBST** uses **the EQL** test. For example:

```
    $ (SUBST 5 2 '(4 2 (3. 2) 4))
    (4 5 (3. 5) 4)
```

---

*Note: The theory of **L** list structures has non-logical symbols **CAR, CDR, CONS, ATOM** and the following axioms [1, p.213]:*

```
    (CAR (CONS X Y))   =  X;
    (CDR (CONS X Y))   =  Y;
    (NOT (ATOM X))     => (CONS (CAR X) (CDR X)) = X;
    (NOT (ATOM (CONS X Y)))
```

---

(1) Mathematical logic in programming: Collection of articles from 1980-1988: Translated from English. - Moscow: Mir, 1991. - 408 p.

In the next step we will look at the input/output functions.

Step 12.
Input/output functions

In this step we will look at the functions that perform input and output of information.

Table 1. **Input/output functions**

Function	**Purpose**
LINELENGTH	Sets the length of the output string.
RDS	Selects a previously opened file for reading.
WRS	Selects a previously opened file for output.
READ	Reads from a file.
READCH	Reads from a file.
PRINT	Outputs the calculated value of the argument and the line feed character to the file.
PRINCIPAL 1	Outputs the value of the argument to the file without outputting a subsequent line feed character.
PRIVATE	Performs a line feed.
SPACES	Outputs the specified number of space characters to a file and returns their number.
System control variable	Controls the output of comment text to a file.

ECHO	

1. The **LINELENGTH** function has one argument, which can be either a numeric atom or a **NIL** atom.

If the argument is a number, the **LINELENGTH** function sets the length of the output string to that number. If the argument is **NIL**, the length of the string remains the same.

In both cases, the value of the **LINELENGTH** function is ***the length of the output string before the function is executed***. The default string length is 79. For example:

```
    $ (LINELENGTH 73)
    79
    $ (LINELENGTH NIL)
    73
    $ (LINELENGTH 80)
    73
```

2. The **RDS** function, whose argument is an S-expression, selects a previously opened file for ***reading***.

The WRS function, whose argument is an S-expression, selects a previously opened file for ***output.***

The RDS (WRS) function with the **NIL** argument switches the input (output) to the standard device. For personal computers, this device is usually ***the keyboard (display)***.

3. Direct reading from a file already selected by the last call to the **RDS** function is performed by the **READ** and **READCH** functions. Both of these functions have no parameters.

The READCH function returns as its value ***the next character*** from the current input file (that is, the file selected for reading by the last call to the **RDS** function).

The READ function returns as its value ***a Lisp expression*** from the current input file (i.e., the file selected for reading by the last call to the **RDS** function).

Let's say the last call to the **RDS** function was **(RDS)**. As soon as the interpreter encounters (**READ**), the calculations are suspended until the user enters some symbol or Lisp expression. Note that **READ "silently waits" for the user to enter an expression. The READ** function "reads" the expression and returns the expression itself as its value, after which the calculations continue.

The above call to the **READ** function had no arguments, but the function has a value that matches the input expression. **READ** is a function in its action, but it has a side effect, which is precisely the input expression. Because of this, **READ** is not a pure function, but ***a pseudo-function***.

If the value read needs to be saved for later use, the **READ** call must be an argument to an assignment function, such as **SETQ**, which will bind the resulting expression to the given atom. For example:

```
    $ (SETQ INPUT (READ))
    (PLUS 2 3)
    $ INPUT
    (PLUS 2 3)
```

The READ function, together with other functions, allows you to read expressions external to the program. From them, you can build new Lisp expressions or entire programs!

The RATOM function returns the atom entered from the console (the rest is ignored).

4. The **PRINT** function outputs to the file already selected for output by the last call to the **WRS** function, the calculated value of the argument and the "line feed" character. For example:

```
   $ (PRINT "Print text to screen!")
   Outputting text to the screen!
```

Like the **READ** function, **PRINT** is a pseudo-function that has both a side effect and a value. The value of the function is the value of its argument, and the side effect is printing that value.

Note that I/O functions are very flexible because they can be used as arguments to other functions. For example:

```
   $ (PLUS (PRINT 2) 3)
   2
   5
```

If we use **PRINT** at the top level, the S-expression is printed ***twice***. The reason for this is that the **PRINT** function (as well as **READ** in a similar situation) is not only used for its action (printing), but also has a value (this value is the value of its argument).

So, if we assign **X** the value (1 2 3) and write **(PRINT X)** at the very top level, the first thing that will happen is that the value of **X will be printed, which is what the PRINT** function itself specifies, i.e. (1 2 3) will be printed, and then the LISP system will do what it always does at the very top level, which is to print the value of the function called, which in this case is the value of **X**, i.e. (1 2 3) will be printed again.

This undesirable phenomenon, of course, disappears when **PRINT** is used normally, i.e. not at the very top level.

The PRINT function is often used temporarily within a program to print intermediate results while debugging the program.

5. The **PRIN1** function outputs to the file already selected for output by the last call to the **WRS** function the value of the argument without subsequently outputting the line feed character.

Thus, the **PRIN1** function acts the same as the **PRINT** function, but the line feed character is not printed to the output file. For example:

```
   $ (PRIN1 1) (PRIN1 2) (PRINT 3)
   1 2 3
```

Both the PRINT function and the **PRIN1** function can output other types of data in addition to atoms and lists, such as ***strings***:

```
   $ (PRIN1 "A B C")
   A B CA B C
```

Outputting information in this form to a file allows the READ function **to** re-read the output expression in a form logically identical to the original.

6. It is often desirable to split the output of expressions into several lines. The line feed can be implemented using the **TERPRI (TERminate PRInting) function. The TERPRI** function has no arguments and returns **NIL** as a value. For example:

```
    (PRIN1 A) (TERPRI) (PRINT B) (PRIN1 C)
   A
   B
   C
```

*<u>Note</u>: The **PRINT** function can be defined using **TERPRI** and **PRIN1** as follows:*

```
   (PRIN1 X) (TERPRI)
```

7. The **SPACES** function outputs the specified number of space characters to the file already selected for output by the last call to the **WRS function and returns their number. Function syntax: (SPACES X)**.

8. The system control variable **ECHO** (not a function!) controls the output of comment text to the file already selected for output by the last call to the **WRS** function. For example:

```
   (SETQ ECHO T)
   % The text will be displayed on the display screen %
   (SETQ ECHO NIL)
   % The text will NOT be displayed on the display screen %
```

In the next step we will get acquainted with the **COND** control structure.

Step 13.
COND control structure

In this step we will look at one of the main control structures **COND**.

The COND function ("CONDition") is the main means of branching calculations.

The structure of a conditional expression is as follows:

```
   (COND (P₁ A₁)
         (P₂ A₂ )
     ...
         (Pɴ Aɴ)
   )
```

Here **(P_1 A_1), ...,(P_N A_N)** are the arguments of the **COND** function. The value of the **COND** function is determined as follows.

1. Expressions **P_i**, acting as predicates, are evaluated sequentially from left to right (top to bottom) until an expression is encountered whose value is not **NIL** (note that strict **T** is not required!).

2. The expression **A_i** corresponding to this predicate **P_i** is evaluated, and the resulting value is returned as the value of the **COND** function.

3. If all **P_i (i=1,2, ...,N)** return **NIL**, then the value of the **COND** function will be **NIL**.

It is recommended to use the symbol **T** as the last **P_N, and the corresponding resulting expression will always be evaluated in cases where no other condition is met. Although T** is not actually necessary, since the same result will be obtained without it.

Good programming practice advocates using **T**, because the presence of **T** makes it clear that the last clause should work if all else fails.

Example 1.

```
(COND ( (EQ N 0) 1 )
      ( (EQ N 1) N )
      ( (EQ N 2) (TIMES M M) )
)
```

Example 2. Check whether a given element **X** is contained in a given list **LST**, and if not, add this element to the list.

```
    (COND ( (MEMBER X LST) T )
          (  T  (CONS X LST) )
    )
```

(см. MEMBER, CONS)

Example 3. Statistics show that in the texts of real programs, calls to the **CAR** function occur on average 10% (sometimes up to 40%) more often than calls to the **CDR** function. The reason for this preference can be explained, for example, by the fact that the **F** function, which works with lists, often has the following structure:

```
(COND  ( (NULL L) NIL)
       ( (P (CAR L)) (G (CAR L)) )
       (       T     (F (CDR L)) ) )
)
```

<u>Note</u>: 1. The conditional clause may lack the resulting expression A_i *or may contain a new function **COND** in its place:*

```
 (COND (P1 A11)
...
    (Pi)
...
    ( PkA1AN2... ANK )
 )
```

*If the condition does not correspond to a resulting expression, then the result of the **COND** function, if the predicate is true, is the value of the predicate itself.*

*If the condition matches several result expressions, then if it is true, the result expressions are evaluated sequentially from left to right, and the result of the **COND** function is the value **of the last** expression in the sequence. It is clear that any expressions between the first and last elements in the **COND function list should be used only for side effects, since they cannot affect the value of COND** in any way.*

*2. The **COND** function is often criticized for its abundance of parentheses. Therefore, other, in various respects more natural, conditional statements are used in different **LISP systems.***

*For example, one of the positive features of the **muLISP** dialect is a construct called **built-in COND**. Using built-in **COND**, you can significantly reduce the number of records when organizing branching in a program. The record:*

```
 (COND (Pred1 Expr1.1 Expr1.2 ... Expr1.A)
    (Pred2 Expr2.1 Expr2.2 ... Expr2.B)
            ...            ...
    ( T    ExprN.1 ExprN.2 ... ExprN.K)
 )
```

can be shortened like this:

```
 (Pred1 Expr1.1 Expr1.2 ... Expr1.A)
 (Pred2 Expr2.1 Expr2.2 ... Expr2.B)
         ...     ...         ...
 (      ExprN.1 ExprN.2 ... ExprN.K)
```

In the next step we will talk about user functions.

Step 14.
User functions. Computed functions

At this point we start looking at user functions. In particular, we will talk about the **LAMBDA** function.

As a result of almost half a century of programming practice, a very effective method of compiling programs was developed, in which the task is divided into subtasks, and those, in turn, are divided into elementary functions, a certain combination of which implements the solution of the subtasks and, accordingly, the entire task. The process of dividing into elementary functions ends when each of the obtained functions performs a certain logically separate operation and the result of the work of this function can serve as an input for another function. Such a division leads to the fact that the task is represented by a certain minimal set of functions, which is coded in a language adequate to the nature of the task.

The problem of choosing subfunctions when developing a main function is an age-old problem of good program structuring. Sometimes standard subfunctions reveal themselves, but more often, a good selection of special-purpose subfunctions can greatly simplify the structure of a set of functions as a whole.

Computable functions.

The definition of functions and their evaluation in the **LISP** language is based on ***Church's lambda calculus.*** The lambda expression, borrowed by the language from the lambda calculus, is an important mechanism in practical programming.

In Church's lambda calculus, the function is written as follows:

```
    lambda (x1,x2, ..., xN).fN .
```

In **LISP,** a lambda expression has the form:

```
    (LAMBDA (X1 X2 ... XN) FN)
```

The LAMBDA function is intended to define "nameless" functions and is called ***a computable function*** (not to be confused with the concept of a computable function in the theory of algorithms!).

The first argument of the function being evaluated - **(X1 X2... XN)** is a list (possibly empty!). It is called ***a lambda list***. **X1, X2, ..., XN** are called ***formal parameters***.

The formality of the parameters means that they can be replaced with any other symbols without affecting the calculations defined by the function. This is what is hidden behind the lambda notation.

The second argument of the **LAMBDA - FN** function is called ***the body***. It is an arbitrary expression whose value can be calculated by the **LISP** interpreter. Thus:

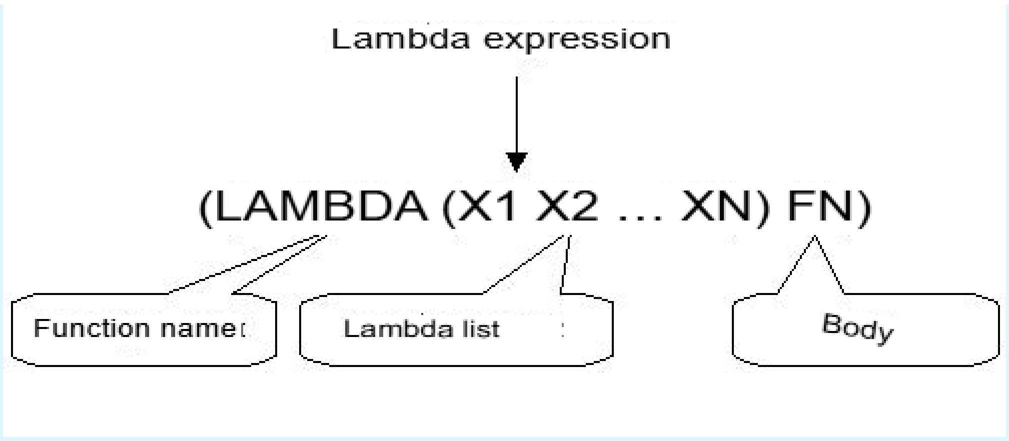


Fig. 1. Lambda expression type

To apply a lambda function to arguments, you need to put the lambda expression in place of the function name in the function call:

```
  (LAMBDA (X1 X2 ... XN) FN) A1 A2 ... AN)
```

Here **Ai (i=1,2, ..., N) are LISP** expressions that define ***the actual parameters***. This form of call is called ***a lambda call***.

The evaluation of a lambda call (the application of a lambda expression to the actual parameters) is done in two stages. First, the values of the actual parameters are evaluated and the corresponding formal parameters are bound to the resulting values. This stage is called ***parameter binding***.

At the next stage, taking into account the new links, the body of the lambda expression is calculated, and the resulting value is returned as the value of the lambda call. After the calculation is complete, the formal parameters are returned the links they had before the lambda call was calculated.

This whole process is called ***lambda transformation (lambda conversion)***.

Here are some examples of lambda transformations:

```
    $ ((LAMBDA (NUM) (PLUS NUM 1)) 45)
    46
    $ ((LAMBDA (M N) (COND ((LESSP M N) M) (T N))) 233 123)
    23
    $ (LAMBDA NIL (PLUS 3 5))
    8
    $ (LAMBDA () (PLUS 3 5))
    8
    $ ((LAMBDA (X) (EQ X NIL)) NIL)
    T
```

A lambda expression is an "unnamed" function that disappears immediately after lambda conversion. It is difficult to use again because it cannot be called by name, although the lambda call is available as a list object.

"Unnamed" functions are used, for example, when passing a function as an argument to another function or when forming a function as a result of calculations, in other words, when ***synthesizing programs***.

In the next step we will talk about non-computable user functions.

Step 15.
User functions. Non-computable functions

In this step we will analyze the non-computable user functions.

Using the macro facilities offered by the latest versions **of muLISP** (85 and 87) is one of the most efficient ways to implement complex programs. With the availability of macro facilities, some functions in the language can be defined as macro functions. Such a definition actually sets the law of preliminary construction of the function body immediately before the interpretation phase.

Unfortunately, in the “younger” versions of **muLISP** (81 and 83) macro facilities are absent. Therefore, they have to be modeled using the so-called ***non-computable functions*** or **NLAMBDA *functions***.

NLAMBDA function (“No-spread LAMBDA”) is a special type of lambda expression that provides passing of function parameters by ***name***.

The interpretation of non-computable functions is performed in two stages.

At the first stage, which we will call ***macro expansion***, the lambda definition of the function is formed depending on the current context, at the second stage the interpretation of the created lambda expression is carried out.

NLAMBDA expression has the form:

```
    (NLAMBDA (X1 X2 … XN) FN)
```

The first argument of the **NLAMBDA function - (X1 X2 ... XN)** is a lambda list (possibly empty!), **X1, X2, ..., XN are *formal parameters***.

The second argument of the **NLAMBDA function (FN)** is ***the body***. It is an arbitrary expression whose value can be calculated by the **LISP** interpreter. Thus:

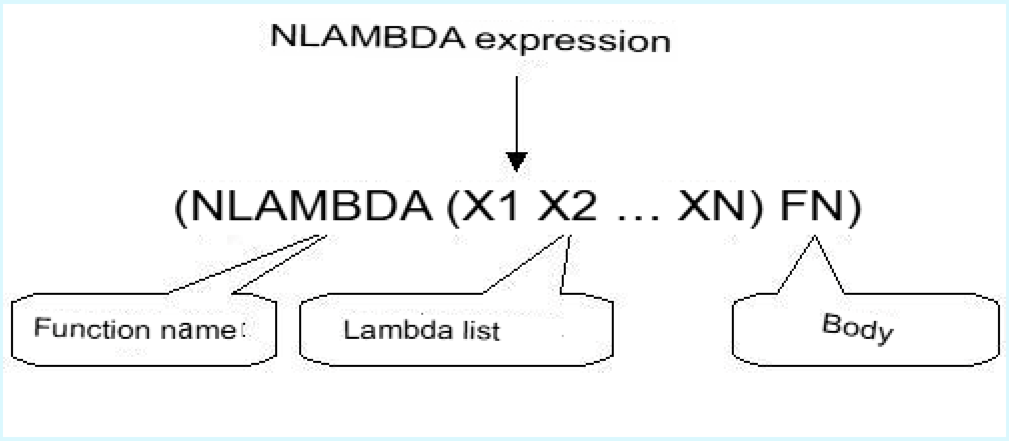


Fig. 1. NLAMBDA expression view

To apply an **NLAMBDA** function to arguments, you need to put an **NLAMBDA** expression in the function call instead of the function name (this will form ***a lambda call***):

```
    (NLAMBDA (X1 X2 ... XN) FN) A1 A2 ... AN)
```

Here **Ai (i = 1, 2, ..., N) are LISP** language expressions defining ***the actual parameters***.

The evaluation **of an NLAMBDA** call (the application of **an NLAMBDA** expression to the actual parameters) is performed as follows: the formal parameters are bound to the corresponding actual parameters. This step is called ***parameter binding***.

At the next stage, taking into account the formed connections, the body of the lambda expression is calculated, and the resulting value is returned as the value of the lambda call. After the calculation is complete, the formal parameters are returned the connections they had before the lambda call was calculated.

This entire process is called **NLAMBDA *transformation***.

Conclusion: If you do not need to calculate the values of the function arguments, then define such a function using **NLAMBDA** expression. For example:

```
$ ((NLAMBDA (X) (CDR X)) (1 (TIMES 1 2) 4))
((TIMES 1 2) 4))
$ ((NLAMBDA (X) (CAR X)) ((TIMES 1 2) (PLUS 1 4)))
(TIMES 1 2)
$ (((NLAMBDA (X) (CAR X)) (PLUS TIMES)) 2 3)
(PLUS 2 3)
```

*Note: Using non-computable functions **(pseudo-macros)** it is easy to **define new syntactic forms**. This is necessary, for example, when adding new control structures, data types to the language, or when implementing an interpreter for a new problem-oriented language. In particular, using non-computable functions facilitates the construction of **a language with a Lisp-like structure** that has a syntax that is more convenient for the user.*

Note, however, that excessive use of non-computable functions makes programs difficult to read and understand.

The next step will be about named functions.

Step 16.
User functions. Named functions (PUTD, GETD and MOVD functions)

In this step we will look at how you can create named functions.

Using the lambda expression apparatus in **LISP,** you can define named functions. For this purpose, you need ***a function constructor*** that binds an atom (function name) to a lambda expression. The result of the binding is called **a LAMBDA** or **NLAMBDA *function definition*. PUTD** and DEFUN act as such constructors.

PUTD, GETD and MOVD functions.

The PUTD function, which has two arguments, binds the atom that is the first argument to **a LAMBDA** or **NLAMBDA** expression that is the second argument, and returns as its value the name of the function being defined. For example:

```
$ (PUTD FOO (LAMBDA (X Y) (CONS X Y)))
FOO
```

Now, having defined a new function **FOO**, we can access it like this:

```
$ (FOO A B)
(A. B)
```

Note that the main purpose of the **PUTD** function is not only to return the function name, but also to associate it with some definition.

To access definitions, there is a **GETD** function with one argument (the function name), which returns the lambda definition of the given function. For example:

```
$ (GETD FOO)
(LAMBDA (X Y) (CONS X Y))
```

Finally, the **MOVD** function assigns the functional value associated with the first atom to the second atom. For example:

```
$ (MOVD FOO BOO)
(LAMBDA (X Y) (CONS X Y))
```

---

*Note: In **muLISP-85, the GETD, PUTD**, and **MOVD** functions are slightly different from those in **muLISP-81**, and a new **REMD** function and several control variables are introduced.*

---

1. If **SYMBOL** is a user-defined function, then the function **(GETD SYMBOL)** decompiles and returns the definition of **SYMBOL** as a list.

If **SYMBOL** is ***a simple function*** or a special form, the function **(GETD SYMBOL)** returns the physical memory address of the function template in machine language.

If **SYMBOL** is not a special form or function, the function **(GETD SYMBOL)** returns **NIL**.

If **FLAG** is not **NIL**, the function **(GETD SYNBOL FLAG)** returns:

- **SPECIAL** if **SYMBOL** is a special form,
- **LAMBDA** if **SYMBOL** is a computation function,
- **NLAMBDA** if **SYMBOL** is not a computation function,
- **MACRO** if **SYMBOL** is a macro function,
- **NIL** if **SYMBOL** is undefined.

2. The **(PUTD SYMBOL DEFINITION)** function compiles the **DEFINITION** definition into D code, modifies the function definition element **SYMBOL** to point to the compiled D code, and returns **SYMBOL**. For example:

```
$ (PUTD 'SQUARE '(LAMBDA (NUM) (* NUM NUM)))
SQUARE
$ (SQUARE 5)
25
```

Non-atom objects (constants) in a function definition are not compiled, but are stored as a linked list. Therefore, unlike earlier versions, the **muLISP-85** interpreter does not have to decompile such constants every time they are encountered, and constants may contain dotted pairs with **CDR** elements that are not **NIL** atoms.

The PUTD control variable controls the compression of D code at compile time.

The MACROEXPAND control variable controls macro expansion at compile time.

To communicate with user-defined machine language templates, the **PUTD** function is called with **DEFINITION** equal to the physical memory addresses of those templates.

3. The function **(MOVD SYMBOL1 SYMBOL2)** transfers the function definition from **SYMBOL1** to **SYMBOL2** and returns **SYMBOL2**.

The examples show how **MOVD can be used to assign different mnemonic names to some LISP** selector functions without affecting their execution efficiency.

```
$ (MOVD 'CAR 'FIRST)
FIRST
$ (MOVD 'CDR 'REST)
REST
$ (MOVD 'CADR 'SECOND)
SECOND
$ (MOVD 'CADDR 'THIRD)
THIRD
$ (MOVD 'CADDDR 'FOURTH)
FOURTH
$ (FIRST '(A B C D))
A
```

4. The **(REMD SYMBOL)** function modifies the function definition element **SYMBOL**, making **SYMBOL** undefined. The **REMD** function returns **SYMBOL**. For example:

```
$ (PUTD 'SWITCH '(LAMBDA (SYM1 SYM2) (CONS SYM2 SYM1)))
$ (SWITCH 'DOG 'CAT)
 (CAT. DOG)
$ (REMD 'SWITCH)
SWITCH
$ (GETD 'SWITCH)
NIL
$ (SWITCH 'DOG 'CAT)
Undefined Function
```

5. When a function definition is compiled, a search is made for existing D-code equivalents for each subexpression in the definition. If such a D-code equivalent is found, it is used instead of further D-code generation, thereby saving memory. This process is called ***condensation***.

If ***the control variable*** **PUTD** is not **NIL**, then the search during condensation is limited to the definition's own D-code. Although this increases the amount of memory used for the D-code, the search area limitation significantly reduces the time required to read and compile files containing a large number of function definitions.

If **PUTD** is **NIL**, the condensation search covers all existing D codes. This reduces the amount of memory required to store a given number of function definitions, but increases the time it takes to compile the function definitions.

When a user-defined function is redefined with **PUTD** or **MOVD**, or its definition is deleted with **REMD**, the D-code used to store the definition is automatically reasserted unless full condensation was performed (unless **PUTD** was **NIL**).

In the next step we will continue our acquaintance with named functions.

Step 17.
User functions. Named functions (DEFUN function)

In this step, we will continue to look at the tools that allow you to create named functions.

You can define a new function and give it a name using the **DEFUN** (**DE** fine **FUN** ction) function.
The **DEFUN** function is called like this:

```
    (DEFUN ATOM LAMBDA expression)
```

The DEFUN function connects an **ATOM** to a **LAMBDA** expression, and **ATOM** begins to represent (name) the computations defined by that lambda expression.

The DEFUN function returns the name of the new function.

Formal parameters of a function are also called ***lexical or static variables***. The links of a static variable are valid only within the function in which they are defined. They cease to be valid in functions called during the calculation, but textually described outside the given function. After the function is calculated, the links of formal parameters created at that time are eliminated and a return to the state that existed before the function was called occurs.

Changes to the values of free variables (i.e. those used in the function but not included in its static variables) that result from a side effect remain in effect after the function has finished executing.

A LISP program consists of many short functions. The functions call each other in a uniform form, which is determined by the use of the lambda mechanism. The body of the lambda expression provides the ability for functions to dynamically call each other or for a function to recursively call itself (directly or indirectly).

The order of function definitions in the program text has no significance from the point of view of the program logic, since the order of calculations in functional programming is not directly expressed. The course of program execution and the time and memory required for it will only become apparent during its execution.

Example 1.

```
  (DEFUN SUMMA (LAMBDA (X Y)
     (MORE X Y)))
  % --------------------- %
  (DEFUN ADD1 (LAMBDA (NUM)
  % Increment function %
     (PLUS NUMBER 1)
  ))
  % --------------------- %
  (DEFUN SUB1 (LAMBDA (NUM)
  % Decrement function %
     (DIFFERENCE NUM 1)
  ))
  % ------------------------ %
  (DEFUN PERCENT (LAMBDA (A B)
  % The function returns A percent of the number B %
     (QUOTENT (TIMES A 100) B)
  ))
  % -------------------- %
  (DEFUN ABS (LAMBDA (NUM)
  % The ABS function returns the absolute value
  of the NUM % argument.
     (COND ( (MINUSP NUM) (MINUS NUM))
           ( T NUMBER )
```

```
      )
   ))
   % -------------------- %
   (DEFUN MAX (LAMBDA (M N)
   % The MAX function returns the larger of two numbers %
      (COND ( (GREATERP M N) M )
            ( T  N )
      )
   ))
   % -------------------------------------------- %
   (DEFUN KPLUS (X Y)    % X and Y are lists of the form (x y) %
   % Complex arithmetic fragment %
      (LIST (PLUS (CAR X) (CAR Y))
            (PLUS (CADR X) (CADR Y)))
   ))
```

Example 2. Definition of a non-computable function **LISTQ** that "prohibits" the computation of its arguments.

```
   (DEFUN LISTQ (NLAMBDA (X Y)
      (LIST X Y)
   ))
```

Test example:

```
   $ (LISTQ ((PLUS 3 4) (TIMES 5 6)))
   ((PLUS 3 4) (TIMES 5 6))
```

Example 3. Let us define the **IF** clause as follows:

```
   (IF CONDITION THEN P ELSE Q)
```

and we describe the corresponding non-computable function:

```
   (DEFUN IF (NLAMBDA (CONDITION THEN P ELSE Q)
      (COND ( (EVAL CONDITION) P )
            ( T  Q )
      )
   ))
```

Test examples:

```
   $ (SETQ X '(THE LOTTERY HAS BEEN THROWN))
   (THE LOTTERY HAS BEEN THROUGH)
   $ (IF (ATOM X) THEN OREL ELSE RESHKA)
   RESHKA
```

We don't check whether the THEN and **ELSE** atoms are specified correctly and in the right places, but such a check could be added quite easily.

*Note 1: As in **Interlisp, extra arguments in the argument list are replaced with NIL** when calling a function. These extra arguments are available for use as local variables within the function.*

*Note 2. In **LISP,** there is no distinction between programs and data: both program and data are translated into an internal representation in the form of linked lists (the names of all built-in **LISP** functions are atoms that are defined by the system at the start of execution, but which otherwise look like any other atom). This allows **LISP** programs to dynamically create other **LISP** programs and then execute them. Thus, if necessary, a data list can be turned into a program list and vice versa.*

*For example, let us represent in graphical notation the list corresponding to the following **LISP** function:*

```
    (LAMBDA (X) (COND ((NULL X) NIL) (T (CAR (CDR X)))))
```

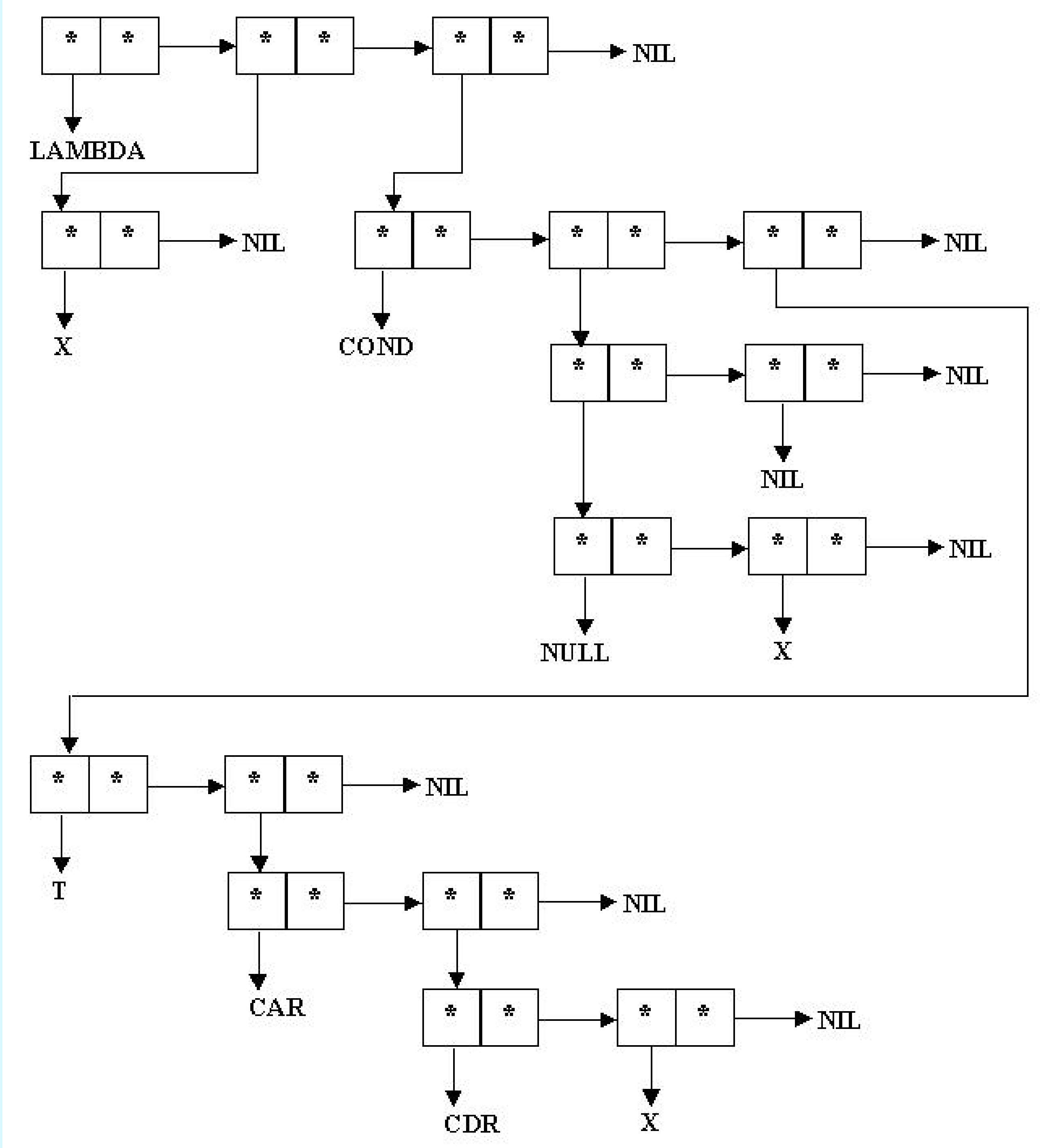


Fig. 1. Graphical representation of the list

Note 3. Initial expressions are used to manipulate function definitions. When a function is defined, the definition is automatically pseudo-compiled into a highly compressed, compact form - the so-called ***D-code****.*

The reverse process of decompiling function definitions into a linked list is done automatically using the ***GETD*** *function to reconstruct the definitions.*

Therefore, both compilation and decompilation ***are invisible to the user****.*

The special form for defining functions **DEFUN** is called in two ways. The first way is more concise and can only be used to define functions. The second way can also be used to define non-computable functions and macros.

The function **(DEFUN SYMBOL ARGLIST FORM1... FORMN)** compiles **ARGLIST** and forms it into D-code, assuming the function type is computed or **LAMBDA**. The **DEFUN** function then modifies the function definition member **SYMBOL** to point to the resulting D-code and returns **SYMBOL**.

If **FTYPE** is a **LAMBDA, NLAMBDA**, or **MACRO** symbol, then **(DEFUN SYMBOL (FTYPE ARGLIST FORM1 ... FORMN))** compiles **ARGLIST** and forms it into D code, assuming that **FTYPE** is a computed, noncomputed, or macro function definition. **DEFUN then modifies the SYMBOL** function element to point to the resulting D code and returns **SYMBOL**.

If **ADDRESS** is zero or a positive integer ***less than 65536***, the function **(DEFUN SYMBOL ADDRESS)** modifies the function member **SYMBOL** to point to **ADDRESS** and returns **SYMBOL**.

The implementation of the function is given below:

```
  (DEFMACRO DEFUN (SYMBOL . BODY)
     ( (ATOM BODY) NULL )
     ( (NUMBERP (CAR BODY))
        (LIST 'PUTD (LIST 'QUOTE SYMBOL) (CAR BODY)) )
     ( (MEMBER (MODEL BODY) '(LAMBDA NLAMBDA MACRO))
        (LIST 'PUTD
              (LIST 'QUOTE SYMBOL) (CONS 'QUOTE BODY)) )
     ( LIST 'PUTD (LIST 'QUOTE SYMBOL)
                 (LIST 'QUOTE (CONS 'LAMBDA BODY)))
  )
```

In the next step we will continue our acquaintance with named functions, in particular, we will consider the creation of macros in the **muLISP-85** system.

Step 18.
User functions. Macros in the muLISP-85 system

In this step, we will finish our discussion of the tools that allow us to create named functions.

It is often useful not to write out the expression to be calculated manually, but to form it using a program. In this case, the calculation of such an expression or part of it can be prevented, if necessary, by a lock (**QUOTE**), for example, for the purpose of transforming expressions. To calculate the expression formed, the programmer can always call the interpreter **EVAL**.

The most natural way to programmatically generate expressions is with ***macros***. Externally, macros are defined and used in the same way as functions, only the way they are calculated differs.

When a macro is called, an expression is first constructed from its arguments, as defined by the macro definition. This expression is then evaluated. Thus, the macro is evaluated in two stages.

Function

```
   (DEFMACRO SYMBOL ARGLIST FORM1 ... FORMN)
```

destructures and compiles the argument list **ARGLIST** and all forms **FORM1, ..., FORMN** (called ***the macro body***) into D-code, ***modifies*** the function definition member **SYMBOL** to a pointer to the resulting D-code, and returns **SYMBOL**.

Examples.

```
(DEFMACRO INCQ1 (SYM N)
   ( (NUMBERP N)
        (LIST 'SETQ SYM (LIST '+ SYM N)) )
   (LIST 'SETQ SYM (LIST 'ADD1 SYM))
)
; ----------------
(DEFMACRO AND1 LST
   ( (NULL LST) )
   ( (NULL (CDR LST)) (CAR LST) )
)
; ---------------
(DEFMACRO OR1 LST
   ( (NULL LST) NIL )
   ( CONS 'COND
          (MAPCAR '(LAMBDA (FORM) (LIST FORM)) LST))
)
```

Unlike the **DEFUN** function, the **DEFMACRO** function provides ***destructuring of* the ARGLIST** argument list. This means that the arguments are assigned to the symbols in **the ARGLIST** based on the linked list structure of **the ARGLIST.**

Example: As an example of ***the difference between a macro and a function,*** consider the implementation of the operation of adding an element to the stack, called **PUSH**, programmed first as a function and then via a macro:

```
(DEFUN PUSH1 (A P)
   (SETQ P (CONS A (EVAL P)))
)
; -------------------
(DEFMACRO PUSH2 (A P)
   (LIST 'SETQ P (LIST 'CONS A P))
)
$ (SETQ S '(B C))
(B C)
$ (PUSH1 'A 'S)              ; in this example, the apostrophe on A is optional
(A B C)
$ S
(B C)
```

Note that when calling the **PUSH1 function,** ***apostrophes*** are used in the arguments. Let's continue testing:

```
$ (PUSH2 D S)
(D B C)
$ S
(D B C)
```

The apostrophe is no longer needed when calling a function!

If **ARGLIST** is a symbol, then the actual arguments in the macro call are bound to symbols. On the other hand, **the CAR** elements of the arguments are recursively bound to **the CAR** elements **of ARGLIST**, and **the CDR** elements of the arguments are bound to **the CDR** elements **of ARGLIST**.

A macro call (macrocall) is a list of forms

```
    (MACRO FORM1 FORM2 ... FORMN),
```

where: **MACRO** is the name of the macro function, **FORM1, FORM2, ..., FORMN** are the actual parameters of the **MACRO** macro.

For example, let's define a macro:

```
  (DEFMACRO SETQQ (X Y)
     (LIST 'SETQ X (LIST 'QUOTE Y))
  )
```

and call it:

```
  $ (SETQQ Q (A S D))
  (A S D)
  $ Q
  (A S D)
```

Normally, macro expansion is generated automatically when a macro call is compiled or executed. However, the following functions allow the user to see how the call is made.

If **FORMA** is ***a macro call***, then the function

```
  (MACROEXPAND-1 FORMA)
```

performs ***macro expansion*** **of FORMA** (implements macro function actions on **FORMA**) and returns the result. Otherwise, **MACROEXPAND-1** returns **FORMA**.

For example, let's define the following macro:

```
  (DEFMACRO DECQ1 (SYM N)
     ( (NUMBERP N)
          (LIST 'SETQ SYM (LIST '- SYM N)) )
     (LIST 'SETQ SYM (LIST 'SUB1 SYM))
  )
```

and let's run a test example:

```
  $ (SETQ A 92)
  92
  $ (DECQ1 A 2)
  90
  $ (MACROEXPAND-1 '(DECQ1 A 2))
  (SETQ A (-A 2))
```

Let's give another example. Let's define ***a recursive macro***:

```
  (DEFMACRO COPY-LISTQ (X)
     (COND ( (NULL X) NIL )
          (  T  (LIST 'CONS (LIST 'QUOTE (CAR X))
                            (LIST 'COPY-LISTQ (CDR X))
                )
          )
     )
  )
```

and let's turn to the **MACROEXPAND-1** function:

```
$ (COPY-LISTQ (A S D))
(A S D)
$ (MACROEXPAND-1 '(COPY-LISTQ (A S D)))
(CONS (QUOTE A) (COPY-LISTQ (S D)))
```

If **FORMA** is ***a macro call***, then the function

```
(MACROEXPAND FORMA)
```

repeatedly expands **FORMA** until the result of the expansion is a macro call.

The MACROEXPAND function returns the result of ***the last expansion***.

Although the **muLISP** interpreter automatically expands macro calls at run time (when a function is evaluated), it is usually more convenient to perform macro expansion only once, at compile time. For macro expansion to occur at compile time, the macro must be defined before the definition of the function containing the macro call, and the **MACROEXPAND** control variable must have a non- **NIL** value.

If the **MACROEXPAND** control variable is not **NIL**, then macro calls are expanded in the same way as by the **MACROEXPAND** function at ***compile*** time (when the function is defined with **DEFUN** or **PUTD**).

If the control variable **MACROEXPAND** is **NIL**, then macro calls are not expanded at compile time.

---

*Note 1: The implementation of **the DEFMACRO, MACROEXPAND-1** and **MACROEXPAND** functions is given below:*

```
(PUTD DEFMACRO 'MACRO BODY
   (POP BODY)
   (LIST 'PUTD
         (LIST 'QUOTE (CAR BODY))
         (LIST 'QUOTE
               (LIST 'MACRO
                     'BODY
                     (CONS (LIST* 'LAMBDA
                                  (Leaves (CADR BODY))
                                  (CDDR BODY)
                           )
                           (Destructure (CADR BODY)
                                        '(CDR BODY))
                     ))))
)
; ------------------
(DEFUN Leaves (TREE)
   ( (ATOM TREE) (LIST TREE) )
   ( (NULL (CDR TREE)) (Leaves (CAR TREE)) )
   ( NCONC (Leaves (CAR TREE)) (Leaves (CDR TREE)))
)
; ----------------------------
(DEFUN Destructure (TREE FORM)
   ( (ATOM TREE) (LIST FORM) )
   ( (NULL (CDR TREE))
        (Destructure (CAR TREE) (LIST 'CAR FORM)) )
   ( NCONC (Destructure (CAR TREE) (LIST 'CAR FORM))
        (Destructure (CDR TREE) (LIST 'CDR FORM)) )
)
; -------------------------
(DEFUN MACROEXPAND-1 (FORM)
   ( (ATOM FORM) FORM )
   ( (NOT (SYMBOLP (CAR FORM))) FORM )
   ( (NOT (EQ 'MACRO (GETD (CAR FORM) T))) FORM )
```

```
      ( APPLY (CAR FORM) FORM )
   )
   ; -------------------------------------
```
Here is the implementation of the MACROEXPAND function:

```
   (DEFUN MACROEXPAND (FORM)
      ( (ATOM FORM) FORM )
      ( (NOT (SYMBOLP (CAR FORM))) FORM )
      ( (NOT (EQ 'MACRO (GETD (CAR FORM) T))) FORM )
      ( MACROEXPAND (APPLY (CAR FORM) FORM) )
   )
```
*Note 2: The **muLISP-85** version of a function definition is a list consisting of three parts:*

- *function type names,*
- *formal parameters and*
- *function bodies.*

The CAR element of a function definition is the name of the function type, i.e. **LAMBDA, NLAMBDA**, or **MACRO**. The function type gives the interpreter information about how to use the function.

A LAMBDA function ***is called a computed function***. When a computed function is called, each of its arguments is evaluated, and only its value is passed to the function itself.

A function of type **NLAMBDA *is called non-computable***. When a non-computable function is called, its arguments are passed to it without being evaluated, just as they are in the calling command. Since using non-computable functions can lead to various conflict situations, it is usually preferable to use macro functions than non-computable ones.

A function of the MACRO type ***is called a macro function***. When a macro function is called, ***the macro definition*** is first used to correctly call the macro without evaluation. The result of such macro expansions is then evaluated and returned as the value of the macro. Macros are controlled by the control variable **MACROEXPAND**.

The CADR element of a function definition in the **muLISP-85** system is:

- or a list of formal arguments,
- or the name of a formal argument.

A function defined with a list of formal arguments (including an empty one) is called ***expanded***. When such a function is called, the actual arguments are associated with the corresponding formal arguments.

A function defined with a formal argument name is called ***undiluted***. When such a function is called, the list of actual arguments is associated with the formal argument name. Therefore, undiluted functions can accept arbitrary actual argument names.

Note that in the **muLISP-85** system, the decision to make a function expanded or not is made independently of the decision to make it computable or not.

CDDR - The element of a function definition in the **muLISP-85** system is a list, which is called ***the function body***.

After the formal arguments of a function are associated with its actual arguments, the implicit function PROGN begins to evaluate the function body. When the function body is evaluated, the formal arguments again take their original values, and the result of evaluating the function body is returned as the function value.

In the next step we will introduce the concept ***of a functional***.

Step 19.
Functional. Interpreter language LISP EVAL

In this step we will get acquainted with the **EVAL** function.

So far we have looked at functions that take non-function expressions as arguments.

An argument whose value is a function is called ***a functional argument*** in functional programming, and a function that has a functional argument is called ***a functional***.

Interpreter language LISP EVAL.

The EVAL function is a function that can evaluate any well-formed **LISP** expression. For example:

```
    $(EVAL(PLUS 2 3))
    5
    $ (EVAL (LIST (READ) (READ) (READ)))
    PLUS
    2
    3
    5
```

The EVAL ***function*** is said ***to define the semantics of Lisp forms***.

An "extra" interpreter call can, for example, remove the effect of blocking the calculation using the **QUOTE** function or allow finding the value of an expression. For example:

```
    $ (EVAL (QUOTE (PLUS 1 2)))
    3
```

QUOTE and **EVAL** act "in mutually opposite directions" and seem to cancel each other's effect.

There is no equivalent for the **EVAL** function in imperative programming languages. Using **EVAL**, we can execute an "operator" that is actually created by our **LISP** program and that may change during the execution of the program.

---

*Note 1. The dialogue with the **LISP** language interpreter at the highest, command level, can be described by a simple cycle:*

```
   (PRINT (QUOTE _))      % Print invitation %
   (PRINT (EVAL (READ))) % Enter expression and %
                          % calculating its value %
   (PRINT (QUOTE _))      % Repeat prompt output %
    ...
```

The interpreter answers the question asked of him and waits for a new task.

*Note 2. Many **LISP** systems have a function **EVAL-QUOTE**, which is similar to **EVAL** except that it has two arguments. The first argument is the name of the function, and the second is a list of its arguments. The arguments of **EVAL-QUOTE** are not evaluated. The function **EVAL-QUOTE** can be defined as:*

```
(EVAL (READ) (READ))
```

*Note 3. It is necessary to note one more feature of the **LISP** language, which follows from the nature of the organization of data structures and programs and the mechanism of their interpretation. The language makes it easy to implement the tasks of **automatic program synthesis**. It allows one to form definitions of other functions using some functions, programmatically analyze and edit these definitions as S-expressions, and then, using functions of the **EVAL** type, execute them.*

*Note 4. Let us describe **the semantics of the EVAL** function for the **muLISP-85** version.*

*If **OBJECT** is an atom, then the function **(EVAL OBJECT)** returns the contents of the **OBJECT** value cell. Since references to newly selected atoms are automatic, atoms "evaluate themselves" until they are bound to other values.*

*If **the CAR** element of **OBJECT** is the name of a function being evaluated or **a LAMBDA**, then the function **(EVAL OBJECT)** evaluates each element **of the CDR** part **of OBJECT** and adds **the CAR** element **of OBJECT** to the result list.*

*If **the CAR** element **of OBJECT** is the name of a non-evaluable function, then the function **(EVAL OBJECT)** appends **the CAR** element **of OBJECT** to **the CDR** element **of OBJECT** without evaluating the latter.*

*If **the CAR** element **of OBJECT** is the name of a macro function, then the function **(EVAL OBJECT)** recursively computes the result of adding **the CAR** element of **OBJECT** to its **CDR** element.*

*If **the CAR** element **of OBJECT** is not a function, then the function **(EVAL OBJECT)** **causes an** UNDEFINED error and generates an error trap.*

In the next step we introduce the concept ***of an applicative functional***.

Step 20.
Applicative functionals. APPLY function

In this step we will introduce the concept of ***applicative functionals***.

One of the main types of functionals are functions that allow calling other functions, in other words, applying a functional argument to its parameters. Such functionals are called ***applicative (applying) functionals***.

Applying functionals are related to the universal LISP function **EVAL**. While **EVAL** evaluates an arbitrary expression, applying functionals evaluate the value of a function call.

The APPLY function is a function of two arguments, of which the first argument is a function that is applied to the elements of the list that make up the second argument of the **APPLY function: (APPLY FN List)**. For example:

```
   $ (APPLY PLUS (2 3))
   5
```

Using **APPLY** gives greater flexibility than calling the function directly: using the same **APPLY** function, you can perform different calculations depending on the function argument. For example:

```
   $ (APPLY (CAR (PLUS - * /)) (2 3))
   5
```

Example 1. Define a functional **(APL-APPLY FX)** that applies each function **Fi** of a list **F = (F1 F2...FN)** to the corresponding element **Xi** of a list **X = (X1 X2... XN)** and returns a list formed from the results.

```
(DEFUN APL-APPLY (LAMBDA (F X)     % F and X - lists %
   (COND ( (NULL F) NIL )
         (    T     (CONS (APPLY (CAR F) (LIST (CAR X)))
                          (APL-APPLY (CDR F) (CDR X))) )
   )
))
```

Example 2. The main user interface with **muLISP** is provided by the read-evaluate-print loop function **DRIVER**:

```
(DEFUN DRIVER (LAMBDA (RDS WRS)
   (LOOP
      (TERPRI) (PRIN1 (QUOTE "> "))
      (PRINT (APPLY (READ) (READ) (TERPRI)))
   )
))
(DRIVER)
```

In the next step we will continue to study ***applicative functionals***, in particular we will consider applicative functionals in **muLISP-85**.

Step 21.
Applicative functionals in muLISP-85

In this step we will list the applicative functionals in **muLISP-85**.

Let us describe the action of applicative functionals in the **muLISP-85** version.

Table 1. **Applicative functionals of muLISP-85**

Function	**Purpose**
(APPLY FUNCTION ARG1 ARG2 ... ARGLIST)	Passes the actual arguments **(ARG1, ARG2, ...** plus ARGLIST elements**) to FUNCTION** in machine language and returns the result of the function.
(FUNCALL FUNCTION ARG1 ARG2 ... ARGN)	Performs **FUNCTION** operations on arguments **ARG1, ARG2, ..., ARGN** and returns the result.
(CONSTANTP OBJECT)	Check if an argument is a constant.
(UNDEFINED SYMBOL FORM1 FORM2 ... FORMN)	Triggers an interrupt on the error "Undefined function".

1. If **FUNCTION** is the machine language name of an action, then the function **(APPLY FUNCTION ARG1 ARG2 ...ARGLIST)** passes the actual arguments **(ARG1, ARG2, ...** plus the elements of **ARGLIST) to FUNCTION** in machine language and returns the result of the function.

If **FUNCTION** is the name of a user-defined function or the body **of a LAMBDA**, then the **APPLY** function binds the formal arguments of **FUNCTION** to the actual arguments **(ARG1, ARG2, ...** plus the elements of **ARGLIST)**, evaluates the function body, restores the original value of the formal arguments, and returns the value of the evaluation of the function body. For example:

```
   $ (APPLY 'CONS '(A (B C D)))
   (A B C D)
   $ (APPLY '(LAMBDA (N) (* N N)) '(5))
   25
```

If **FUNCTION** is not a function name or a **LAMBDA** function body, **APPLY** generates an "Undefined function" trap.

2. If the function **(FUNCALL FUNCTION ARG1 ARG2 ... ARGN)** performs the **FUNCTION** operations on the arguments **ARG1, ARG2, ..., ARGN** and returns the result. The **FUNCTION function must be the name of a computed or non-computed function, or the body of a LAMBDA** function. For example:

```
   $ (FUNCALL 'CONS 'A '(B C D))
   (A B C D)
   $ (FUNCALL '(LAMBDA (N) (* N N)) 5)
   25
```

If FUNCTION **is** the name of a macro or an undefined function, an "Undefined function" error occurs.

Here is the function code:

```
   (DEFUN FUNCALL (FUNC ARG1 ARG2 ... ARGN)
      (APPLY FUNC ARG1 ARG2 ... ARGN NIL)
   )
```

3. If **OBJECT** is a constant, then the function **(CONSTANTP OBJECT)** returns **T**, otherwise **NIL**. For example:

```
   $ (CONSTANTP ())     $ (CONSTANTP -237.6)
   T                    T
   $ (CONSTANTP 'T)     $ (CONSTANTP '(QUOTE (A B C)))
   NIL                  T
```

Note that **OBJECT** is a constant if and only if the function **(EVAL OBJECT)** returns **OBJECT**.

The NIL symbol, numbers, and lists whose **CAR** element is the **QUOTE** symbol are treated as ***constants*** in **muLISP**.

The function code looks like this:

```
   (DEFUN CONSTANTP (OBJ)
      ( (NULL OBJ) )
      ( (NUMBERP OBJ) )
      ( (ATOM OBJ) NIL )
      ( EQ (CAR OBJ) 'QUOTE )
   )
```

4. The function **(UNDEFINED SYMBOL FORM1 FORM2 ... FORMN)** initiates an interrupt on the error "Undefined function" by calling the function:

```
   (BREAK (LIST SYMBOL FORM1 FORM2 ... FORMN)
          '"Undefined Function")
```

This error handling function is called when an attempt is made to evaluate a form whose **CAR** element is a character that does not have a function definition. This helps the user determine the type of error.

Code functions obvious:

```
   (DEFUN UNDEFINED LST
      (BREAK LST '"Undefined Function") )
```

The FUNCALL *functionality is similar in its action to the* ***APPLY*** *functionality, but it accepts arguments for the called function not as a list, but "individually". The syntax of the functionality:*

```
                (FUNCALL FN X1 X2...XN)
```

For example:

```
   $ (FUNCALL '+ '2 '3)
   5
```

Define the **FUNCALL functionality in terms of the APPLY** functionality in the **muLISP-81** version.

In the next step we will begin to study ***mapping functionals***.

Step 22.
Display functionals (general information)

In this step we will give general information about the display functionalities.

An important class of functionals are ***mapping functionals*** (**MAP *functions***). **MAP functions** are functions that map a source list to a new list or generate a side effect associated with the source list.

The names of **MAP** functions begin with the prefix **MAP**, and their calls look like this:

```
    (MAP* FN L1 L2 ... LN)
```

Here:

- ***** - a specific sequence of characters;
- **L1, ...,LN** - lists;
- **FN** is a function of **N** arguments.

Typically, the **MAP** function is applied to a single argument list, i.e. **FN** is a function of one argument:

```
   (MAP* FN LIST)
```

There are two main types of **MAP** functions.

Some of them use the functional argument **FN** in such a way that its arguments will be successively the elements of the argument list.

Others apply the functional argument **FN** to successive **CDRs** of the list argument. The result of these repeated computations is a list consisting of the results of successive applications of the function.

In all cases, the number of argument lists must match the number of arguments of the function used for calculations.

In the next step we will continue to study ***mapping functionals***, in particular, we will consider the **MAPCAR** function.

Step 23.
Display functionals. MAPCAR function

In this step we will look at the **MAPCAR** function.

Let's consider two functions first.

Example 1.

```
(DEFUN SUMADD (LAMBDA (LST)
% A function that converts a list of integers into a list
  each element of which is one greater than
  the corresponding element %
     ( (NULL LST) NIL )
     ( CONS (PLUS 1 (CAR LST)) (SUMADD (CDR LST)) )
))
% ------------------------ %
(DEFUN TANUSHA (LAMBDA (LST)
% A function that converts a list of integers into a list,
each element of which is the remainder
of the corresponding element divided by 2 %
     ( (NULL LST) NIL )
     ( CONS (REMAINDER (CAR LST) 2) (TANUSHA (CDR LST)) )
))
```

It is obvious that there is something common in the structure of these functions.

A natural question arises: is generalization possible?

Below we will construct a **MAPCAR** function for the **muLISP-81** dialect, with the help of which the second problem of interest to us is solved as follows:

```
$ (MAPCAR REMAINDER (2 4 6))
(0 0 0)
```

The call to the **MAPCAR** function looks like this:

```
(MAPCAR FN (X1 X2... XN))
```

The value of the MAPCAR function is computed by applying the **FN** function to successive elements **Xi** of the list that is the second argument **of MAPCAR**.

The value of the functional is returned as a list constructed from the results of calls to **the MAPCAR** functional argument.

The function code for **muLISP-81** is quite simple:

```
(DEFUN MAPCAR (LAMBDA (FUN LST)
   (COND ( (NULL LST) NIL )
         ( CONS (FUN (CAR LST))
                (MAPCAR FUN (CDR LST)) )
   )
))
```

Note here that the calculation of **the MAPCAR** functional can be performed ***in parallel*** and does not depend on the order of processing the list elements.

Example 2.

```
$ (MAPCAR ATOM (A B C))
(T T T)
$ (MAPCAR (LAMBDA (Y) (LIST (PRINT Y))) (A B C))
A
B
C
((A) (B) (C))
$ (PUTD PARA1 (LAMBDA (X) (LIST X 1)))
MONEY1
$ (MAPCAR PARA1 X)
((A 1) (B 1) (C 1))
$ (MAPCAR NUMBERP (1 2 OVER 3))
(T T NIL T)
```

Example 3. Given a list containing positive integers, construct a function that returns a list consisting of factorials of each element of the original list.

```
(DEFUN DEMO (LAMBDA (FUN LST)
   (PRIN1 "Result of MAPCAR action: ")
   (MAPCAR FUN LST)
))
% ------------------- %
(DEFUN FACT (LAMBDA (X)
   (COND ( (ZEROP X) 1 )
         ( T (TIMES X (FACT (DIFFERENCE X 1))) )
   )
))
% --------------------------- %
(DEFUN MAPCAR (LAMBDA (FUN LST)
   (COND ( (NULL LST) NIL )
         (   T    (CONS (FUN (CAR LST))
                        (MAPCAR FUN (CDR LST))) )
   )
))
```

Test example:

```
$ (DEMO FACT (2 3 4))
(2 6 24)
```

Example 4. The direct product of two numeric sets can be defined through two nested loops, which can be expressed using the composition of two nested calls to the **MAPCAR** functional:

```
(DEFUN DECART (LAMBDA (X Y))
 % X and Y are lists %
   (MAPCAR (QUOTE (LAMBDA (X)
      (MAPCAR (QUOTE (LAMBDA (Y) (LIST X Y))) Y))) X)
))
% ----------------------------- %
(DEFUN MAPCAR (LAMBDA (FUN LST)
    (COND ( (NULL LST) NIL )
          ( T  (CONS (FUN (CAR LST))
                     (MAPCAR FUN (CDR LST))) )
    )
))
```

Test example:

```
$ (DECART (A B C) (1 2 3))
(((A 1) (A 2) (A 3)) ((B 1) (B 2) (B 3)) ((C 1) (C 2)
(C 3)))
```

*Note 1: Note that calling the **MAPCAR** function as follows:*

```
(MAPCAR FN (X1 X2... XN))
```

*is equivalent to calling the **LIST** function:*

```
(LIST (FN X1) (FN X2)... (FN XN))
```

*Note 2. The function **(MAPCAR FN LIST1...LISTN)** performs the operations prescribed by the function **FN** on **the CAR** elements of the lists **LIST1, ...,LISTN**, then on the **CADR** elements of each list, and so on until each list is exhausted. The function **MAPCAR** returns a list of results. For example:*

```
$ (MAPCAR '*'(2 3 5 7) '(2 4 6 8))
(4 12 30 56)
```

*The function code in the **muLISP-85** system is as follows:*

```
(DEFUN MAPCAR (FUNC LST1 LST2)
   ( (OR (NULL LST1) (NULL LST2)) NIL )
   (CONS (FUNCALL FUNC (CAR LST1) (CAR LST2))
         (MAPCAR FUNC (CDR LST1) (CDR LST2)))
)
```

In the next step we will continue to study ***mapping functionals***, in particular, we will consider the **MAPLIST** function.

Step 24.
Display functionals. MAPLIST function

In this step we will look at the **MAPLIST** function.

The MAPLIST function works like the **MAPCAR** function, but it operates on successive **CDRs** (starting from the source list) of the list rather than on the elements of the list.

Here is the function code for **muLISP-81**:

```
(DEFUN MAPLIST (LAMBDA (FUN LST)
   (COND ( (NULL LST) NIL )
         (   T   (CONS (FUN LST)
                       (MAPLIST FUN (CDR LST))) )
   )
))
```

Test example:

```
$ (MAPLIST (LAMBDA (Y) Y) (A B C))
((A B C) (B C) (C))
```

An example.

```
(DEFUN DEMO (LAMBDA NIL)
   (PRIN1 "Result of MAPLIST action: ")
   (MAP FROM SUM (1 2 3))
))
% ---------------------- %
(DEFUN SUMMA (LAMBDA (LST)
% The function returns the sum of the elements of the list LST %
   (COND ( (NULL LST) 0)
           (  T  (PLUS (CAR LST) (SUMMA (CDR LST))) )
      )
   ))
   % ---------------------------- %
   (DEFUN MAPLIST (LAMBDA (FUN LST)
      (COND ( (NULL LST) NIL )
            (   T   (CONS (FUN LST)
                          (MAPLIST FUN (CDR LST))) )
      )
   ))
```

Test example:

```
   $ (DEMO)
   (6 5 3)
```

In **muLISP-85,** the function **(MAPLIST FN LST1...LSTN)** performs the operations specified by the **FN** function on the elements of the lists **LIST1, ...,LISTN**, then on **the CDR** elements of each list, and so on until ***each list*** is exhausted. The **MAPLIST** function returns a list of results. For example:

```
   $ (MAPLIST 'REVERSE '(A B C D))
   ((D C B A) (D C B) (D C) (D))
```

The function code in the **muLISP-85** system looks like this:

```
   (DEFUN MAPLIST (FUNC LST1 LST2)
      ( (OR (NULL LST1) (NULL LST2)) NIL )
      (CONS (FUNCALL FUNC LST1 LST2)
            (MAPLIST FUNC (CDR LST1) (CDR LST2)))
   )
```

The MAPCAR and **MAPLIST** functionals are used to program special types of loops because they can be used to express repetitive calculations concisely.

In the next step we will continue to study ***mapping functionals***, in particular, we will consider the **MAPC** function.

Step 25.
Display functionals. MAPC function

In this step we will look at the **MAPC** function.

The syntax of the MAPC function in **muLISP-81** is:

```
   (MAPC FN LST)
```

Where:

- **FN** is a function name or **LAMBDA** expression,
- **LST** - list.

The MAPC function applies **FN** sequentially to the elements of the list, i.e. to **(CAR LST), (CAR (CDR LST))** and so on.

Code functions in **muLISP-81** like this:

```
   (DEFUN MAPC (LAMBDA (FUN LST)
      (LOOP
         ( (NULL LST) NIL )
         (PRINT (FUN (POP LST)))
      )
   ))
```

An example.

```
   (DEFUN DEMO (LAMBDA (LST)
      (PRIN1 "MAPC action result: ") (MAPC PRIMER LST)
   ))
   % ----------------------- %
   (DEFUN PRIMER (LAMBDA (LST)
      (PLUS (CAR LST) 1)
   ))
   % ------------------------- %
   (DEFUN MAPC (LAMBDA (FUN LST)
      (LOOP
         ( (NULL LST) NIL )
         (PRINT (FUN (POP LST)))
      )
   ))
```

In **muLISP-85,** the function **(MAPC FN LST11...LSTN)** performs the operations specified by the function **FN** on **the CAR** elements of the lists **LST1, ...,LSTN**, then on **the CDR** elements of each list, and so on until ***each list*** is exhausted. The function **MAPC** returns the list **LST1**. For example:

```
   $ (MAPC 'PRIN1 '(DOG CAT COW))
   DOGCATCOW
```

Note that the function will return **(DOG CAT COW)**.

Code functions in **muLISP-85** like this:

```
   (DEFUN MAPC (FUNC LST1 LST2)
      ( (OR (NULL LST1) (NULL LST2)) LST1 )
      ( FUNCALL FUNC (CAR LST1) (CAR LST2) )
      ( MAPC FUNC (CDR LST1) (CDR LST2) )
      LST1
   )
```

Display functionals do not increase the computational power of **LISP**, but they are certainly useful visual aids. In many cases, they can be used to significantly reduce the notation compared to repeated computations expressed by recursion or iteration.

*Note: There is one more function in the **muLISP-85 version: MAPL**. The function **(MAPL FUNC LIST1... LISTN)** performs the operations defined by **FUNC** on the elements **of LIST1, ...,LISTN**, then on the **CDR** elements of each list, and so on, until each list is exhausted. **MAPL** returns **LIST1**. For example:*

```
   $ (MAPL 'PRIN1 '(A B C))
   (A B C) (B C) (C)
```

Here is the code for this functionality:

```
   (DEFUN MAPL (FUNC LST1 LST2)
      ( (OR (NULL LST1) (NULL LST2)) LST1 )
      ( FUNCALL FUNC LST1 LST2 )
      ( MAPL FUNC (CDR LST1) (CDR LST2) )
      LST1
   )
```

In the next step we will move on to ***the unifying functionalities***.

Step 26.
Unifying functionals

In this step we will introduce the concept of unifying functionals.

Function:

```
   (MAPCAN FUNC LIST1... LISTN)
```

performs the operations specified in the **FUNC** function on **the CAR** elements of the lists **LIST1, ...,LISTN**, then on the **CADR** elements of each list, and so on until each list is exhausted. The **MAPCAN function *"links"* its results using the NCONC** function.

MAPCAN functionality is useful for selecting unwanted elements from a list. The following example shows how to remove negative numbers from a list:

```
   $ (MAPCAN '(LAMBDA (N) ( (MINUSP N)) (LIST N))
   '(3 -7 4 0 -5 1))
   (3 4 0 1)
```

Function:

```
   (MAPCON FUNC LIST1... LISTN)
```

performs the operations specified in the **FUNC** function on **the CAR** elements of the lists **LIST1, ...,LISTN**, then on the **CDR** elements of each list, and so on until each list is exhausted. The **MAPCAN function *"links"*** its results using the **NCONC** function. For example:

```
   $ (MAPCON 'REVERSE '(A B C D))
   (D C B A D C B D C D)
```

Note that the **MAPCON** function can easily ***"loop" through*** lists, so use this function with caution.

---

*Note: The **MAPCAN** and **MAPCON** functions are analogs of the **MAPCAR** and **MAPLIST** functions. The difference is that **MAPCAN** and **MAPCON** do not build a new list from the results using **LIST**, but combine the lists that are the results into one list using the **NCONC** function.*

***The MAPCAN** and **MAPCON** functions can be defined using the **MAPCAR** and **MAPLIST** functions as follows:*

```
 (MAPCAN FN x1 x2...xN) <--->
                      (APPLY (NCONC (MAPCAR FN x1 x2...xN)))
 (MAPCON FN x1 x2...xN) <--->
                      (APPLY (NCONC (MAPLIST FN x1 x2...xN)))
```

In the next step we will look at ***the function scheduling functionalities***.

Step 27.
Functionalities of Function Planning

In this step, we will analyze what the function planning functionalities are.

The scheduling functions discussed in this section execute test functions on the elements of one or more lists until a "termination criterion" or the end of one of the lists is encountered. ***The termination criterion*** is based on the truth value returned by the test function. Note that scheduling functions are ***predicates***.

Scheduling functionality is available only in the **muLISP-85** and **muLISP-87** dialects.

Table 1. **Functionalities of planning functions**

Function	**Purpose**
(SOME TEST LIST1...LISTN)	Performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** objects of each list, and so on until the test returns a value other than **NIL** or the end of the list is encountered.
(NOTANY TEST LIST1...LISTN)	Performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** objects of each list, and so on until the test returns a value other than **NIL** or the end of the list is encountered.
(EVERY TEST LIST1...LISTN)	Performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** objects of each list, and so on until the test returns **NIL** or the end of the list is encountered.
(NOTEVERY TEST LIST1...LISTN)	Performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** objects of each list, and so on until the test returns **NIL** or the end of the list is encountered.
REDUCE FUNC LIST	Converts the elements of **the LIST** list to a simple value using the **FUNC** function, a function of two arguments.

1. The function **(SOME TEST LIST1... LISTN)** performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** objects of each list, and so on until the test returns a value different from **NIL** or the end of the list is encountered.

If the test returns a value other than **NIL**, the **SOME** function returns that value; if the end of the list is reached, the **SOME** function returns **NIL**. For example:

```
    $ (SOME 'EQL '(DOG CAT COW) '(COW CAT DOG))
    T
    $ (SOME 'PLUSP (LIST 0 -3 (+4 -6)))
    NIL
```

Here is the function code:

```
    (DEFUN SOME (TST LST1 LST2)
       (LOOP
           ( (OR (NULL LST1) (NULL LST2)) NIL )
           ( (FUNCALL TST (POP LST1) (POP LST2)) )
       )
    )
```

2. The function **(NOTANY TEST LIST1 ... LISTN)** performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** objects of each list, and so on until the test returns a value different from **NIL** or the end of the list is encountered.

If the test returns a value other than **NIL**, the **NOTANY** function returns **NIL**, and if the end of the list is reached, the **NOTANY** function returns **T**. For example:

```
$ (NOTANY 'EQL '(DOG CAT COW) '(COW CAT DOG))
NIL
$ (NOTANY 'ODDP (LIST 0 (+3 3) 7/2))
T
```

Here is the function code:

```
(DEFUN NOTANY (TST LST1 LST2)
   (NOT (SOME TST LST1 LST2))
)
```

3. The function **(EVERY TEST LIST1... LISTN)** performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** objects of each list, and so on until the test returns **NIL** or the end of the list is encountered.

If the test returns **NIL**, the **EVERY** function returns **NIL**, and if the end of the list is reached, the **EVERY** function returns **T**. For example:

```
$ (EVERY 'EQL '(DOG CAT COW) '(DOG CAT PIG))
NIL
$ (EVERY 'ODDP (LIST 3 5 7 11 13))
T
```

Here is the function code:

```
(DEFUN EVERY (TST LST1 LST2)
   (LOOP
       ( (OR (NULL LST1) (NULL LST2)) NIL )
       ( (NOT (FUNCALL TST (POP LST1) (POP LST2))) T )
)
```

4. The function **(NOTEVERY TEST LIST1... LISTN)** performs the actions of the predicate **TEST** on **the CAR** elements of the lists **LIST1, ..., LISTN**, then on the **CADR** elements of each list, and so on until the test returns **NIL** or the end of the list is encountered.

If the test returns **NIL**, the **NOTEVERY** function returns **T**, and if the end of the list is reached, the **NOTEVERY** function returns **NIL**. For example:

```
$ (NOTEVERY 'EQL '(DOG CAT COW) '(DOG CAT PIG))
T
$ (NOTEVERY 'STRING< '(BILL JACK JOE) '(BUD JIM SUE))
NIL
```

Code functions obvious:

```
(DEFUN NOTEVERY (TST LST1 LST2)
   (NOT (EVERY TST LST1 LST2))
)
```

5. The function **(REDUCE FUNC LIST)** sequentially transforms the elements of **the LIST** list to a simple value using the **FUNC** function - a function of two arguments.

The transformation is performed from left to right.

The function **(REDUCE FUNC LIST INITIAL)** transforms the elements of the **LIST** list, taking **INITIAL** as the initial value. For example:

```
$ (REDUCE 'CONS '(A B C D))  $ (REDUCE '* '(2 3 5 7) -2)
(((A. B). C). D)            -420
$ (REDUCE '* '(2 3 5 7))     $ (REDUCE '* NIL)
210                          1
$ (REDUCE '* '(1 2 3 4 5))   $ (REDUCE 'LIST '(1 2 3 4 5))
120   % 5!=120               ((((1 2) 3) 4) 5)
```

```
    $ (REDUCE 'LIST '(1 2 3 4 5) '(A B C))
    ((((((A B C) 1) 2) 3) 4) 5)
```

The next step will provide examples of solving some problems using the theory presented.

Step 28.
Practical lesson #1. The simplest data types of the LISP language. Interactive mode. The COND function

In this step we will give several problems and their solutions.

Let us quote from T. Kuhn's monograph, which reflects our approach to conducting laboratory classes:

"One can regularly hear from the former (students) that they have read through a chapter of the textbook, understood thoroughly everything that it contains, but nevertheless have difficulty in solving a number of problems offered at the end of the chapter. Usually these difficulties are resolved in the same way, as happened in the history of science. The student, with or without the help of his instructor, finds a way to liken the problem to those he has already encountered. Having noticed such a similarity, having grasped the analogy between two or more different problems, the student begins to interpret the symbols and himself bring them into conformity with nature in ways that have previously proven their effectiveness. Let us say that the formula of the law **F = ma** functioned as a kind of tool, informing the student about what analogies exist for it, designating a kind of gestalt through the prism of which a given situation should be considered.

The ability to see in all the diversity of situations something similar between them... is, I think, the main thing that the student acquires by solving sample problems with pencil and paper or in a well-equipped laboratory. After he has completed a certain number of such problems or exercises (this number can vary greatly depending on his individual characteristics), he looks at the situation as a scientist, with the same eyes as the other members of the group in the given specialty. For him, these situations will no longer be the same as those with which he dealt when he began to carry out educational tasks. Now he has a way of seeing that has been tested by time and approved by the scientific group" [1, p. 246-247].

The simplest data types of the LISP language.

1. Use graphical notation to represent the following S-expressions:

- (A . (CAT . (EATS . MICE)));
- (A . (CAT . EATS) . MICE);
- ((A . CAT) . (EATS . MICE));
- (A . ((B . (C . NIL)) . ((D . (E . NIL)) . NIL))) ;
- ((A . NIL) . NIL).

2. Are the following S-expressions equivalent:

- (NIL) and (NIL . NIL);
- ((A B)) and ((A . B) . NIL) ;
- ((A B)) and (NIL . (A B)) ?

3. Convert the following point records into list records:

- (A . (B . (C . NIL)));
- ((A . NIL) . NIL) ;
- (NIL . (A . NIL)) ;
- (A . ((B . (C . NIL)) . ((D . (E . NIL)) . NIL))) ;

- ((A . (B . NIL)). (C . ((D . NIL). (E . NIL)))) ;
- (A . (B . ((C . (D . ((E . NIL) . (NIL))) . NIL)))).

4. Convert the following list entries into dot notations using graphical notation:

- (W (X));
- ((W) X);
- (NIL NIL NIL) ;
- (V (W) X (Y Z));
- ((V W) (X Y) Z);
- (((V) W X) Y Z).

Working in interactive mode. COND function.

1. Which function whose name begins with C, ends with R, and contains the letters A and D (for example, **CAADR**) returns:

- (3 4) when applied to (1 2 (3 4))?;
- (3 4), when applied to (1 2 3 4) ?
- (6 3), when applied to ((4 (6 3) 8) 7) ?
- 12, when applied to (5 ((12) 23 34)) ?
- (12), when applied to (5 ((12) 23 34)) ?

2. Which function, when applied to a list of more than three elements, returns:

- a list consisting of the third element of the original list;
- the third element of the original list;
- the sum of the first and second elements of the list;
- the product of the first, second, and third elements of the list.

3. For each of the following conditions, define a conditional expression that returns the value **T** if the condition is satisfied and **NIL** otherwise:

- the first element of the list LST is 12;
- the first element of the list LST is an atom;
- list LST has less than three elements;
- the second element of the list LST is greater than the third;
- list LST is empty;
- list LST has at most four elements.

4. Write a conditional expression that:

- returns NIL if the absolute value of the difference between its first and second arguments X and Y is less than its third argument Z, and T otherwise;
- returns NIL if L is an atom and T otherwise;
- returns, for an list LST containing three elements, the first of them that is an atom, or the list (1 2 3 4) if the list contains no elements that are atoms;
- returns T if the argument value is 1, and 0 otherwise.

5. Write a conditional expression that tests whether its argument is:

- a list containing two elements;
- a list of two atoms;

- a list of three elements;
- a list of three atoms;
- an atom;
- list

6. Let **L1** and **L2** be lists. Write a conditional expression that returns **T** if the first two elements of these lists are respectively equal to each other, and **NIL** otherwise.

7. A numeric list contains 2 elements. Write a conditional expression that returns the larger of the elements in the list.

8. A numeric list contains 3 elements. Write a conditional expression that returns the smallest of the elements in the list.

---

[1] Kuhn T. Structures of Scientific Revolutions. - M.: Progress, 1977. - 300 p.

In the next step we will provide the rules for working with the **muLISP-81** programming system.

Step 29.
muLISP-81 programming environment

In this step we will look at the order in which programs are created in **muLISP-81**. This information will be useful when creating programs in later versions.

Let's consider the most important components of the **muLISP-81** programming environment in the order of their use at the stages of program development.

Coding and execution stage.

The program text is first placed into a file using a text editor.

When entering an expression into the **muLISP-81** system, blank lines and spaces are ignored. A space is used to separate function arguments, and multiple spaces can always be used instead of a single space.

You can include comments in the program text, which are enclosed in % symbols. For example:

```
   % This is a comment... %
```

The interpreter is loaded from the **MS-DOS** operating system using the command

```
   C:\>LISP
```

After this, you will see the following on the display screen:

```
   muLISP-80 2.12 (07/09/81)
   Copyright (C) 1981 MICROSOFT
   Licensed from The SOFT WAREHOUSE
   $
```

The "$" symbol is the so-called ***prompt***, by which the interpreter lets you know that it has evaluated the value of the previous expression and is waiting for a new expression.

After the "$" prompt, you can access the following **muLISP-81** functions (Table 1):

Table 1. **muLISP-81 functions**

Function	**Purpose**
(RDS FILENAME EXT)	Loads the program stored in the **FILENAME.EXT** file into the computer's memory.
(SAVE FILENAME)	Saves the current memory state (working environment) in the **FILENAME.SYS file.**
(LOAD FILENAME)	Restores the previously written to disk working environment (functions and variables) from the file **FILENAME.SYS**. The same can be done in the command line: **C:\>LISP FILENAME**.
(SYSTEM)	Return to the operating system.

If the function definitions were syntactically correct and no error message was returned, you can begin testing their operation.

Testing stage.

For testing programs, the muLISP-81 system includes the **TRACE** function. Using ***tracing,*** you can track the evaluation of the function (or functions) being tested in order to localize and correct errors. If a function is marked as traceable, the system informs about the function name and argument values each time the function is entered and accordingly outputs the function value as soon as it is evaluated.

Tracing is enabled by calling:

```
   $ (TRACE)
   NIL
```

Next, we mark the functions we need as "traced":

```
         (TRACE Traced_function_names)
```

The names of the functions being traced are separated by spaces. For example:

```
   $ (TRACE PLUS1 PLUS2)
   NIL
```

After this, the interpreter traces the calculation of the functions **PLUS1** and **PLUS2**.

Tracing can be disabled with the **UNTRACE** directive:

```
   $ (UNTRACE PLUS1)
```

Found errors are corrected in a text editor.

Here are the source codes for the **TRACE** and **UNTRACE** functions.

```
   % File: TRACE.LIB (c) 06/26/80 The Soft Warehouse %
   % The arguments of the TRACE function are the names of the functions that
   which we want to trace. The trace contains the names of the
   called functions and arguments and return value
   after calculations. Only functions defined by
   ned using LAMBDA and NLAMBDA %
   (PROG1 "" (PUTD DEFUN (QUOTE (NLAMBDA (FUNC DEF)
                              (PUTD FUNC DEF) FUNC ))) )
   % --------------------------------------------------- %
   (DEFUN TRACE (LAMBDA LST
      (SETQ INDENT 0)
      (MAPC LST (QUOTE (LAMBDA (FUN BODY FUN#)
         (SETQ BODY (GETD FUN))
         (SETQ FUN# (PACK (LIST FUN #)))
         (MOVD FUN FUN#)
         ((MEMBER (CAR BODY) (QUOTE (LAMBDA NLAMBDA)))
            (PUTD FUN (LIST (CAR BODY) (CADR BODY)
            (LIST EVTRACE FUN (CADR BODY) FUN#) )) )
         (PRIN1 FUN)
      (PRINT " is not a LAMBDA defined function") )) )
   ))
   % --------------------- %
   (DEFUN UNTRACE (LAMBDA LST
   % The UNTRACE function cancels tracing %
      (MAPC LST
          (QUOTE (LAMBDA (FUN FUN#)
                      (SETQ FUN# (PACK (LIST FUN #)))
                      ( (GETD FUN#) (MOVD FUN# FUN)
                                    (MOVD NIL FUN#) )
          ))
      )
   ))
   % ---------------------------------- %
   ( DEFUN EVTRACE ( NLAMBDA ( FUN ARGS FUN #) )
      (PRTARGS FUN ARGS)
      (PRTRSLT FUN (APPLY FUN# (MAKARGS ARGS)))
   ))
   % ---------------------------- %
   (DEFUN PRTARGS (LAMBDA (FUN ARGS)
      (SPACES INDENT)
      (SETQ INDENT (PLUS INDENT 1)) (PRIN1 FUN) (PRIN1 " [")
      ( (NULL ARGS) (PRINT "]") )
      ( LOOP
          ( (ATOM ARGS) (SETQ ARGS (EVAL ARGS))
             ( LOOP
                 (PRIN1 (CAR ARGS)) (SETQ ARGS (CDR ARGS))
                 ( (ATOM ARGS) ) (PRIN1 ", ")
             )
          )
          (PRIN1 (EVAL (CAR ARGS)))
          (SETQ ARGS (CDR ARGS))
          ( (NULL ARGS) )
          (PRIN1 ", ")
      )
      (PRINT "]")
    ))
   % ---------------------------- %
   (DEFUN PRTRSLT (LAMBDA (FUN RSLT)
      (SETQ INDENT (DIFFERENCE INDENT 1)) (SPACES INDENT)
      (PRIN1 FUN) (PRIN1 " = ") (PRINT RSLT) RSLT
   ))
   % ------------------------ %
   (DEFUN MAKARGS (LAMBDA (ARGS)
      ( (NULL ARGS) NIL)
      ( (ATOM ARGS) (EVAL ARGS) )
      (CONS (EVAL (CAR ARGS)) (MAKARGS (CDR ARGS)))
   ))
   % ------------------------- %
   (DEFUN MAPC (LAMBDA (LST FUNC)
      (LOOP
         ( (NULL LST) NIL)
         ( FUNC (CAR LST) )
         ( SETQ LST (CDR LST) )
     )
   ))
   (RDS)
```

Often, when creating program text, you accidentally type a "Russian" symbol on the keyboard, which is very similar to Latin... How can you detect such an error? Here is a useful utility that allows you to do this:

```
(DEFUN LIKE (LST)
 % Constructing a list of those atoms from the list of atoms LST, which
 the Russian letters A, B, E, K, M, H, O, R, S, T, X in their name,
 similar in spelling to Latin letters %
   (COND ( (NULL LST) NIL )
         ( (RUS_LETTER (UNPACK (CAR LST)))
               (CONS (CAR LST) (LIKE (CDR LST))) )
         (  T   (LIKE (CDR LST)) )
   )
)
% -------------------- %
(DEFUN  RUS_LETTER (LST)
% A predicate that returns T if the atom name contains
  one of the following Russian letters:
  A B E K M N O R S T X %
   (COND ( (NULL LST) NIL )
         ( (MEMBER (CAR LST) '(A V E K M N O R S T X)) T)
         (  T   (RUS_LETTER (CDR LST)) )
   )
)
```

Test example:

```
    $ (LIKE '(AB AA AA))
    (AB AA)
```

The first two atoms in the list used Russian letters.

Broadcast stage.

The tested program can be translated using the **muLISP-81** interpreter using the **SAVE** function. The interpreter creates a machine language version of the program that requires less memory and runs significantly faster than the interpreted version. The difference is usually one order of magnitude, but depends, of course, on the program being translated.

In the next step we will provide the rules for working with the **muLISP-85** programming system.

Step 30. muLISP-85 programming system

In this step we will look at the order of creating programs in **muLISP-85**.

Composition of the system.

The muLISP-85 system contains the following files (Table 1):

Table 1. **Composition of the muLISP-85 system**

File	**Purpose**
***COM** files running on the **MS-DOS** operating system:*	
MULISP-85.COM	**muLISP** compiler and interpreter.
COMTOEXE.COM	Utility for converting **COM** file to **EXE** file.
***muLISP** editor and debugger files:*	

EDIT.LSP	Editor source file.
DEBUG.LSP	Debugger source file.
Utility library files:	
GRAPHICS.LSP	muLISP graphical tools.
MOUSE.LSP	Provides an interface between **muLISP** and **the Microsoft Mouse**.
MULISP-83.LSP	**muLISP-83** compatibility file.
INTERLISP.LSP	**INTERLISP** dialect utility functions.
COMMON.LSP	**COMMON LISP** dialect utility functions. The **MACLISP** and **ZETALISP** dialects are closely related to the **COMMON LISP** dialect, so this file is useful for these dialects of the **LISP** language as well.
Educational system files:	
LESSONS.LSP	Main file.
MULISPn.LES	Files with lessons on the **muLISP** dialect.
Demo program files:	
ANIMAL.LSP	Game "Counting animals".
ANIMAL.MEM	Data tree for the game "Counting Animals".
DOCTOR.LSP	Imitation of the actions of a psychotherapist.
EICHIS.LSP	Game "Rowing competitions".
HANOI.LSP	Game "Towers of Hanoi".
METAMIND.LSP	Game "Deciphering the secret code".

Note: If some of the listed files are missing from the muLISP-85 archive, they can be taken from the muLISP-87 archive.

Loading and operating the system.

After the system has loaded, the following message will appear on the display screen:

```
   muLISP-85  5.nn ( mm/dd/yy )
   xxxxxxxxxxx version
   Copyright (c) 1982 to 85  SOFT WAREHOUSE, Inc.
   Licensed  by MICROSOFT corp.
```

Accordingly, the version numbers (5.nn), month (mm), day (dd) and year (yy) appear. The version number and date, together with the serial number, will be included in all system reports.

The xxxxxxxxxx characters in the message indicate the type of computer on which this version **of muLISP** can run.

Below the message, a muLISP *prompt* appears, consisting of a dollar sign ($) followed by a blank line. The prompt means that the system is ready for console input.

You can enter LISP expressions.

After you type an expression and press **Enter, muLISP *evaluates the expression*** (calculates its value) and prints the result at the beginning of a new line. This cycle of interaction repeats over and over until a call to

SYSTEM is made, which terminates **muLISP** and returns control to the operating system. The cyclic principle of **muLISP** is sometimes called **" oval-LISP**. "

Expressions written in literal and/or dotted notation can be entered from the console. The expression is not evaluated until all parentheses are balanced, so you may end up with an expression that spans multiple lines of the display screen. After you press **Enter** for the "last" time, the expression is "read" by the **READ** function, "evaluated" by the **EVAL** function, and the result is printed to the display screen by the **PRINT** function.

Let's demonstrate the main loop with an example:

```
    $ 'DOG
    DOG
    $ (+ 5 -2 4)
    7
    $ (EQUAL 'DOG 'CAT)
    NIL
    $ (member 'dog '(cat cow dog pig))
    (DOG PIG)
```

---

*Note 1: Note that **muLISP** converts lowercase letters to uppercase as they are read. **If the system variable *READ-UPCASE* is NIL, then muLISP does not perform this conversion**.*

*Note 2: The main user interface with **muLISP is provided by the read-evaluate-print** loop function DRIVER.*

*After initializing the data area, **muLISP** enters a top-level computed driver loop. This top-level loop sets the internal **<break-level>** counter to 0, sets the control variables **RDS** and **WRS** to **NIL**, making **the console** both **the current input source (CIS)** and **output stream (COS)**, sets the control variable **READ-CHAR** to **'READ-CHAR**, setting console input to linear editing, clears the linear editing buffer, and recalls the current value of the **DRIVER** variable.*

*In **LISP** it looks like this:*

```
    (LOOP
       (SETQ break-level 0)
       (SETQ RDS NIL)
       (SETQ WRS NIL)
       (SETQ READ-CHAR 'READ-CHAR)
       (CATCH NIL (FUNCALL DRIVER))
    )
```

*The default value of the **DRIVER variable is the DRIVER** function, which is discussed below. Once a new **DRIVER function is defined, the (RETURN)** command terminates the default **DRIVER function, and the top-level driver loop initiates the new DRIVER** function.*

*The example shows how the "read-calculate-print" cycle is implemented for **muLISP**:*

```
    (DEFUN EVAL-QUOTE()
       (CATCH 'RETURN
              (LOOP
                 (CATCH 'DRIVER (PRINC ">")
                        (PRINT (APPLY (READ) (READ))))))
    )
```

```
    $ (SETQ DRIVER 'EVAL-QUOTE)
    EVAL-QUOTE
    $ (RETURN)
    > CONS (DOG (CAT COW PIG))
    (DOG CAT COW PIG)
```

The (DRIVER) *function executes a* ***muLISP*** *read-calculate-print loop. The loop begins by displaying the current interrupt level, if it is nonzero (the* ***DRIVER*** *function is not called directly by a high-level loop).*

Then the dollar symbol is displayed. Finally, an expression is read from ***the current input source (CIS)*** *using the* ***READ*** *function, evaluated using the* ***EVAL*** *function, and the result is sent to* ***the output stream (COS)*** *using the* ***PRINT*** *function.*

When a read-evaluate-print loop is interrupted while reading, evaluating, or printing expressions, a branch is made to ***the DRIVER*** *label. This feature allows functions such as* ***BREAK*** *to interrupt program execution and immediately transfer control to the current loop level.*

This loop continues until the expression ***(RETURN VALUE)*** *is read and evaluated. At this point, the loop terminates and the* ***DRIVER*** *function returns* ***VALUE****.*

For ease of programming, ***the last three expressions*** *read by the loop can be replaced by the variables:* ***+,*** *++ and +++ respectively, and* ***the results of calculating the last three expressions*** *- by the variables:* ****,*** *** and *** respectively. The currently read expression is replaced by the variable "-".*

The example shows how these variables can be used:

```
    $ (CONS 'SAM '(ANN JOE SUE))
    (SAM JOE SUE)
    $ +
    (CONS 'SAM '(ANN JOE SUE))
    $ **
    (SAM ANN JOE SUE)
```

Take a look at the implementation of the ***DRIVER*** *function:*

```
   (DEFUN DRIVER ()
      (CATCH 'RETURN (LOOP (CATCH 'DRIVER
          ( ((ZEROP break-level))
            (PRIN1 break-level) )
          (PRINC "$ ")
          (SETQ - (READ))
          (PSETQ * (UNWIND-PROTECT (EVAL -)
                                  (SETQ +++ ++ ++ + + -))
                 *** ** ** *)
          (PRINT *) )))
   )
```

Editing lines.

The muLISP-85 system contains ***a line editor*** for editing previously entered or currently entered strings. To produce a line of text, type a series of characters and then press the **Enter** key.

Text editing is performed using ***control characters***. Calling up a control character is typing the corresponding letter while simultaneously pressing the **CTRL** key (Table 2):

Table 2. **Control keys**

Keys	**Purpose**
CTRL-A	Shift the word to the left.
CTRL-C	Shift the end of the line to the left.
CTRL-D	Shift character to the right.
CTRL-F	Shift the word to the right.

CTRL-G	Destroy the symbol under the cursor.
CTRL-H	Shift character to the left.
CTRL-I	Shift right to the next tab stop.
CTRL-J	Write a line of text.
CTRL-M	Write a line of text.
CTRL-P	Get rid of the next character entered.
CTRL-R	Shift the end of the line to the right.
CTRL-S	Shift character to the left.
CTRL-T	Destroying the word "under the cursor".
CTRL-V	Switching between insert/delete modes.
CTRL-X	Shift the end of the line to the left.
CTRL-_	Shift character to the left.
CTRL-Y	Shift the end of the line to the left.
BACKSPACE	Shift character to the left.
DELETE	Destroy the character to the left of the cursor.
ESC	Write a line of text.
ENTER	Write a line of text.

If **muLISP** is running on **an IBM PC**, the cursor control keys (arrows, **HOME, END, INSERT,** etc.) on ***the right side of the keyboard*** are also functional.

Reading source and environment files.

The (RDS 'drive:name.type) function returns the character **<drive:name.type>** if the file **<name.type>** is found on the drive located on the device **<drive>** ; otherwise, it returns **NIL**.

If the file name extension is missing, a file with the extension **".LSP"** is searched for. If the device name is missing, the default device is used.

Note that an operating system command can be used to read the source files. For example:

```
   C:\>MULISP ANIMAL.LSP
```

reads the **ANIMAL.LSP** source file from the default device after the **muLISP** interpreter is loaded from the **C** device.

You can create program source files using either the **muLISP-85** system text editor or an "external" text editor that generates ASCII text **files**.

***Typically the last line of the source file is a command* (RDS) *that switches* muLISP input *to the console after the file has been read*.**

The muLISP *environment* is the current state of the system. It consists of all currently active **muLISP** data structures, variable values, and function definitions.

The SAVE command saves the current **muLISP** environment as a **SYS** file, so the environment can be quickly restored at a later time. For example, the command

```
   (SAVE 'C:WHALE)
```

creates **a SYS** file **WHALE.SYS** on device **C**. If successful, the **SAVE** command returns **T**, otherwise **NIL**.

After the SAVE command has completed, the **muLISP** environment can be restored using the **LOAD** command. For example, the command:

```
   (LOAD 'C:WHALE)
```

loads the **WHALE.SYS** file from the **C** device. If **the SYS** file is not found, the **LOAD** command returns **NIL**, otherwise it does not return a variable but starts working with a new environment. Note that after executing the **LOAD** command, the environment is replaced by the one stored in the **SYS** file.

SYS file can be loaded using operating system commands. For example:

```
  A:\>MULISP C:\WHALE
```

loads **the SYS** file **WHALE.SYS** from the **C** device after loading **muLISP** from the default device.

If the source or **SYS** file is not found when loading using the operating system command, the message **"File not found"** is displayed on the display screen.

At any time ***during program execution***, a user-initiated interrupt system from the console can stop program execution and return control to the console.

A console interrupt is initiated by pressing the **ESC** key on the console keyboard. When a console interrupt occurs, the console screen displays the following message:

```
  Console Interrupt Break: NIL
```

Following this, a prompt appears on the next line in the form of interruption options. The user can then decide for themselves how to continue working.

---

*Note 1. The current **muLISP** environment consists of all variable values, function definitions, and property values. These values and definitions can be entered directly from the console, read from a source file, or loaded from a **SYS** file. Environment functions allow the **muLISP** environment to be loaded and saved as compact files - memory images. Such a file is usually created only after the program has been debugged to the end. Loading memory image files is faster than reading such source files.*

If **type** is a string other than "COM", the function **(SAVE drive:name.type)** saves the current **muLISP** environment as a binary **SYS** file **named name.type** on **the drive** device.

If **type** is missing, **the SYS** file is assigned the type **"SYS"**. If drive is missing, the **SYS** file is saved on the current device.

The function **(SAVE drive:name.COM)** saves the **muLISP** interpreter without the pseudocode compiler, as well as the current **muLISP** environment, as an executable **COM** file **name.COM** on **the drive** device. For example:

```
   $ (SAVE 'C:WHALE)
```

```
      ; Saves the environment to a file named WHALE.SYS
      ; on drive C:
   $ (SAVE 'WHALE.COM)
      ; Saves the environment to a file named WHALE.COM
      ; on the default drive
```

If **drive:** is missing, **the COM** file is saved to the current device. By modifying a flag in the **muLISP** main page, it is possible to save the pseudocode compiler in **the COM** files created by **SAVE**.

Before saving the environment as **a SYS** or **COM** file, the **SAVE** function automatically "compacts" all active data structures so that the resulting file has a minimum size. Consequently, the size of the memory image file increases with the size of the active data structures.

Since **COM** files are larger than **SYS** files, their pseudocode compilers are invalid, and they cannot be loaded into a running **muLISP** system, it is better to use **SYS** files rather than **COM** files to preserve the **muLISP** environment (at least during program development).

If the **COM** file created by the **SAVE** function is less than 64K (65536 bytes), it can be executed as an **MS-DOS** command. If **the COM** file is larger than 64K, it can be converted to an **EXE** file using the **muLISP utility COMTOEXE.COM**.

The command to call this utility is as follows:

```
   drive1: COMTOEXE drive2: filename.COM drive3:     ,

where: drive1 is the device containing COMTOEXE.COM,
     drive2 - the device containing filename.COM, a
     drive3 - the device where the file named filename.EXE is located.
```

Note that any missing device is replaced by the current one.

The function **(LOAD drive:name.type)** replaces the current **muLISP** environment with the environment that existed when the file **drive:name.type** was created.

If **type is missing from the LOAD function argument,.sys** is assumed by default. If **drive:** is missing, the **SYS** file is loaded onto the current device.

The LOAD function does not return any value if **the SYS** file is successfully loaded, but begins executing the function associated with the **DRIVER** variable in the environment that has already been loaded. For example:

```
   $ (load 'C:WHALE) ; Loads the WHALE.SYS file
                     ; from the C drive:
```

If the **SYS** file is not found, the **LOAD** function returns **NIL**.

SYS files can also be loaded along with **MULISP.COM** from the operating system.

System debugger.

The muLISP *Debugging Package* is a collection of algorithms that provide assistance to the user in designing and debugging **muLISP programs. It contains *tracing, interrupt, and statistics*** facilities.

Tracing tools for each function provide information on the display screen about the progress of the function's calculation.

Interrupt facilities stop the evaluation of functions at critical points, allowing the user to perform environment checking.

Statistics tools help determine the speed or depth of critical sections in large programs.

The muLISP debugging facilities are located in the file **DEBUG.LSP**, which can be loaded with the command: **(RDS 'DEBUG)**.

Let the functions **APP** and **REV** be defined as follows:

```
(DEFUN APP (LST1 LST2)
   ( (NULL LST1) LST2 )
   ( CONS (CAR LST1) (APP (CDR LST1) LST2) )
)
; --------------
(DEFUN REV (LST)
   ( (NULL LST) NIL )
   ( APP (REV (CDR LST)) (LIST (CAR LST)) )
)
```

The APP and **REV** functions are equivalent to the **muLISP-85** functions
APPEND and **REVERSE** respectively, however **APP** and **REV** are less efficient and have a maximum of two arguments.

Function tracing is performed using the undrilled **TRACE** function. It is called with arguments representing one or more functions to be traced. For example, the command:

```
(TRACE 'APP 'REV)
```

traces the **APP** and **REV** functions. If you then enter the command:

```
(REV '(A B))
```

then on the display screen you will see the following route:

```
1| REV [LST: (A B)]
2|  REV [LST: (B)]
3|   REV [LST: NIL]
3|   REV = NIL
3|   APP [LST1:NIL,LST2: (B)]
3|   APP = (B)
2| REV = (B)
2|  APP [LST1: (B),LST2: (A)]
3|   APP [LST1: NIL,LST2: (A)]
3|   APP = (A)
2|  APP = (B A)
1| REV = (BA)
(B A)
```

If the trace is being drawn too quickly, you can use the **CTRL-S** key combination to temporarily stop the trace.

Using commands:

```
(SETQ ECHO T)
(WRS 'drive:name.type)
```

you can direct the output trace to a file **named name.type** on **the drive:** device. Once the trace is finished, call the **(WRS)** function, which closes the file. You can then view the file at any time using a text editor.

To eliminate large amounts of unnecessary information, you can select the moment when the trace function will perform the output.

Function:

```
   (SETQ TRACE NIL)
```

delays trace output for all functions until the **TRACE** function is redefined to a value other than **NIL**.

Function:

```
   (REMFLAG 'TRACE 'name)
```

where ***name*** is the name of the function being traced, delays output of the trace for that function until a call to the function is encountered:

```
   (FLAG 'TRACE 'name)
```

Trace output for all functions may be delayed based on the number of calls to the function being traced.

If the **MINCALL** control variable has a positive integer value, then the entire trace output is delayed until the number of function calls is greater than or equal to this value.

In the same way, the trace output for any one particular function can be delayed based on the number of calls.

The function:

```
   (PUT 'MINCALL 'NAME N)

     where: NAME is the name of the function being traced
            N is a positive integer
```

delays the output of a function trace until it has been called **N** times.

Trace output for all functions can be delayed based on the nesting level of the function call.

If the **MINLEVEL** control variable has a positive integer value, then all trace output is delayed until the function nesting level is greater than or equal to this value.

Similarly, if **MAXLEVEL** is a positive integer, trace output is delayed until the function nesting level is less than or equal to that value.

Similarly, based on the nesting level, the trace output for any one particular function may be delayed.

The function:

```
   (PUT 'MINLEVEL 'NAME 'N)
     where: NAME is the name of the function being traced,
            N is a positive integer,
```

delays the output of the function trace until its nesting level is greater than or equal to **N**.

Similarly, the function:

```
   (PUT 'MAXLEVEL 'NAME N)
```

delays the output of the trace of the function named **NAME** until its nesting level is less than or equal to **N**.

By setting **MAXLEVEL** to 1, you can trace only the top-level function call.

For example, the functions:

```
   (PUT 'MINLEVEL 'APP2)
   (PUT 'MAXLEVEL 'REV1)
   (REV '(A B C))
```

perform the following output:

```
   1| REV [LST: (A B C)]
   4|    APP [LST1: NIL,LST2: (B)]
   4|    APP = (B)
   3|   APP [LST1: (B),LST2: (A)]
   4|    APP [LST1: NIL,LST2: (A)]
   4|    APP = (A)
   3|   APP = (B A)
   1| REV = (CBA)
   (C B A)
```

However, even when the trace output has been delayed in one of the above ways, the trace can be displayed on the screen from the so-called trace ***history***. The debug package automatically saves the trace "history" for subsequent restoration of the trace output during such delays. The "history" is saved in a fixed-length buffer; the buffer length is equal to the value of the control variable **HISTLEN**. By default, the value of **HISTLEN** is 15.

If after the previous actions you access the function

```
   (HISTORY)
```

then the following route will appear on the screen:

```
   4|  APP [LST1: NIL,LST2: (C)]
   4|   APP = (C)
   3|  REV = (C)
   3|  APP [LST1: (C),LST2: (B)]
   4|   APP [LST1: NIL,LST2: (B)]
   4|   APP = (B)
   3|  APP = (C B)
   2| REV = (CB)
   2| APP [LST1: (C B),LST2: (A)]
   3|  APP [LST1: (B),LST2: (A)]
   4|   APP [LST1: NIL,LST2: (A)]
   4|   APP = (A)
   3|   APP = (B A)
   2|  APP = (C B A)
   1| REV = (CBA)
```

The CLEAR function is used to disable tracing and interrupt a function. For example, the function:

```
   (CLEAR 'APP 'REV)
```

Disables APP and REV tracing.

If the traced function is interrupted, the function call level counter **LEVELCOUNT** will not be reset to zero. In this case, the function:

```
 (CLEAR)
```

LEVELCOUNT is reset to zero and other related functions are performed.

After the interrupt function is called, the computation is suspended and the **muLISP BREAK** function is called. This allows you to check the contents of the environment at the interrupt point (the values of the variables), display a backtrace - a diagram of the nesting of function calls - on the screen, and also make any changes to the environment. You can "force" the interrupt function to perform interrupts only at certain nesting levels, or only after a certain number of function calls, etc.

The BRK function is used to interrupt functions. It is called with one or more functions as arguments. For example,

```
   (BRK 'REV 'APP)
```

interrupts **the APP** and **REV** functions.

After calling the function:

```
   (REV '(A B C))
```

an interrupt will be executed and the following message will be displayed:

```
   Break-point: REV [LST: (A B C)]
   Continue, Break, Abort, Top-level, Restart, System ?
```

If you want to check the current **muLISP** environment, choose option "B". You can also use the **HISTORY** function.

If you want to see the function call stack at the breakpoint, use the function:

```
   (BACKTRACE)
```

The BACKTRACE function displays the names of all functions currently being evaluated, along with their arguments, starting with the top-level call. Only those functions that were traced or referenced when the interrupt occurred will be included in the backtrace.

To exit the interrupt, call the function:

```
   (RETURN)
```

The calculation of the interrupted function will resume.

If you do not want the function to be evaluated further, replace **NIL** in the previous command with the name of the expression to be evaluated.

If you want to abort the computation entirely and return control to the higher-level **muLISP** driver, call the **RETURN** function:

```
   (RETURN (THROW))
```

The CLEAR function described above disables interruption of functions and restores their original definitions.

The user can choose the moment when the interruptible function will actually be interrupted.

Function:

```
   (SETQ BRK NIL)
```

disables all interrupts until **BRK** returns values other than **NIL**.

Function:

```
   (REMFLAG 'BRK 'NAME)
```

where **NAME** is the name of the function to interrupt, disables interruption of that function until the command returns:

```
    (FLAG 'BRK 'name)
```

The interruption of all functions can be delayed based on the number of calls to the interrupted functions.

If the value of the **MINCALL** control variable is a positive integer, then all interrupts are delayed until the number of function calls is greater than or equal to this value. Similarly, the interrupt of a particular function can be delayed based on the number of calls to that function.

The function:

```
   (PUT 'MINCALL 'NAME N)

     where: NAME is the name of the interrupted function,
            N is a positive integer,
```

delays interruption of the given function until it has been called **N** times.

Termination of functions can be delayed based on the nesting level of the function call. If the value of the control variable **BRKLEVEL** is a positive integer, functions are not terminated until the nesting level of the function equals that value. Similarly, termination of a given function can be delayed based on ***the recursive nesting level of a particular function.***

The function:

```
   (PUT 'BRKLEVEL 'NAME N)

     where: NAME is the name of the interrupted function,
            N is a positive integer,
```

delays the termination of the given function until its recursive nesting level is **N**.

For example, using the functions

```
   (REMFLAG 'BRK 'REV)
   (PUT 'BRKLEVEL 'APP 3)
   (REV '(A B C))
```

the following interrupt will be executed:

```
   Break-point: APP [LST1: NIL, LST2: (A)]
   Continue, Break, Abort, Top-level, Restart, System?
```

The (HISTORY) function will display the trace on the display screen:

```
   2|  REV [LST: (B C)]
   3|   REV [LST: (C)]
   4|    REV [LST: NIL]
   4|    REV = NIL
   4|    APP [LST1: NIL, LST2: (C)]
   4|    APP = (C)
   3|   REV = (C)
   3|   APP [LST1: (C), LST2: (B)]
   4|    APP [LST1: NIL, LST2: (B)]
   4|    APP = (B)
   3|   APP = (C B)
   2| REV = (CB)
   2|  APP [LST1: (C B), LST2: (A)]
   3|   APP [LST1: (B), LST2: (A)]
   4|    APP [LST1: NIL, LST2: (A)]    ,
```

and the function **(BACKTRACE)** will return the following:

```
   REV [LST: (A B C)]
   APP [LST1: (C B), LST2: (A)]
   APP [LST1: (B), LST2: (A)]
   APP [LST1: NIL, LST2: (A)]
```

Once the computation is complete, the user can determine the number of calls to each specific function being debugged.

The value of the CALLCOUNT variable is the total number of calls to each function marked for debugging. The call count for each specific function is stored in the property list of the **CALLCOUNT** variable.

The function:

```
   (CDR 'CALLCOUNT)
```

prints to the display screen all functions marked by the debugger, along with call counters for each of these functions.

Once the calculation is complete, the user can also define ***the maximum call level (depth)*** for each specific function marked for debugging.

The value of the MAXLEVELCOUNT control variable is equal to the maximum call level for the functions being debugged. The maximum recursion level for each specific function is stored in the property list of the **MAXLEVELCOUNT** variable.

Thus, the function:

```
   (CDR 'MAXLEVELCOUNT)
```

displays all marked functions on the display screen along with their maximum recursion level.

The user can also determine ***the current call level (depth)*** for each specific function marked for debugging.

The value of the LEVELCOUNT control variable is equal to the current call level for the marked functions. The current recursion level for specific functions is stored in the property list of the **LEVELCOUNT** variable.

Function

```
   (CDR 'LEVELCOUNT)
```

prints to the display screen all marked functions along with their current recursion levels.

---

*Note: The **(BREAK OBJECT MESSAGE)** function pauses program execution and prints to the console the **MESSAGE** symbol, the string **"Break",** and **OBJECT** indicating the reason for the interruption.*

*If **MESSAGE** is **NIL** or absent, this string is not evaluated. The following prompt string is given in the form of interrupt options:*

```
   Continue, Break, Abort, Top-level, Restart, System?
```

*The system then waits for the user to select one of the options by specifying the first letter of its name **(C, B, A, T or S)**. Note that these options are listed in order of increasing effectiveness:*

- ***Continue*** *- **OBJECT** is returned as the interrupt value, and execution continues as before;*
- ***Break*** *- temporarily suspends execution of a break. **READ-CHAR** is set to **'READ-CHAR** so that console input will be performed in a line-editing manner; the **BREAK** variable is set to **OBJECT**, and an internal counter, **BREAK-LEVEL** (break level), is incremented. The **DRIVER** function is called to allow the user to check what environment is being provided when the break occurs. The **DRIVER** function displays **BREAK-LEVEL** with the driver prompt as a reminder that a break is in progress.*

To exit the break and return to the calling program, follow the break prompt by typing:

```
   (RETURN FORM)
```

*where **FORM** is the expression to be evaluated, and press **Enter**.*

*For example, to evaluate and return **the OBJECT** passed in through a call to **BREAK** as the value of the **BREAK** function, type:*

```
   (RETURN (EVAL BREAK))
```

*On exit from the driver, **BREAK-LEVEL is decremented; the READCH** and **BREAK** functions are restored to their original values; the value of the expression entered following **RETURN** is returned as the break value.*

- ***Abort*** *- interrupts the calculation during the stop process and transfers control to the last of the previously marked **DRIVERs** (usually to the "read-calculate-print" cycle).*
- ***Top-level*** *- interrupts the computation during the stop process and transfers control to the last of the previously marked **NILs** (usually to the top-level driver cycle).*
- ***Restart*** *- closes all open files, discards the current **muLISP** environment, and initiates a new **muLISP** system.*
- ***System*** *- Closes all open files, terminates **muLISP**, and returns control to the operating system.*

If after a few seconds the user has not selected an option, a short message is generated informing the user that an interruption has occurred.

*Whenever **muLISP** encounters an error condition, **BREAK** is called with an appropriate error message, allowing the user to make some decision.*

***The BREAK** function can be redefined by the user to meet other requirements. By introducing **BREAK** calls into function definitions, the interrupt package provides an excellent debugging mechanism.*

In the next step we will look at the features of the **muLISP-85** text editor.

Step 31.
muLISP-85 System Editor

In this step we will look at the capabilities of the **muLISP-85** system editor.

The resident editor **muLISP** significantly reduces the time for program development due to the interactive structure **of LISP**. Using the screen-oriented editor **muLISP**, you can create programs, test them and fully debug them in the **muLISP** environment. In addition, the editor executes certain commands of list structures, performs the selection of parameters of the flickering image and automatic selection of new lines. All this makes the process of program development more creative and productive.

If the memory capacity is insufficient, the source file generated by the **muLISP** editor can be read into the **muLISP** system without the editor, so the memory is not occupied by application programs. In the current version of the editor, the edited text is completely written to memory. This imposes a limitation on the maximum size of source files that can be edited by the **muLISP editor - *50 KB***.

Loading and saving the editor.

The muLISP editor is provided by the system as a source file **EDIT.LSP**. This file can be read and saved as a **SYS** file. The easiest way to load the **EDIT.LSP** file is to enter the command:

```
             MULISP EDIT.LSP
```

This command loads **muLISP, prints the standard muLISP** message to the display screen, and then proceeds to read **EDIT.LSP**. After a few seconds of typing, the system asks whether the user wants to use a **WorldStar-like** or **an Emacs-like** editor. Unless the user has a different opinion, it is recommended to select the **WordStar-like** option.

Once the dollar sign prompt appears, enter the command:

```
               (SAVE EDIT)
```

saving the editor as the last loaded **SYS** file named **EDIT.SYS**.

Once you have saved the **EDIT.SYS** file, during further work with **muLISP** you can use the command to load and run the editor:

```
               MULISP EDIT
```

If **muLISP** has already been loaded, enter the command:

```
               (LOAD EDIT)
```

followed by the dollar sign; the command will ensure that the editor is loaded and begins working. However, it should be noted that the **LOAD** command causes the current **muLISP** environment to be overwritten, including all definitions of functions, properties, etc.

If you need to load the editor into an existing environment, run the command:

```
               (RDS EDIT)
```

Main menu options.

The muLISP editor clears the screen, draws a frame, and displays the main menu:

```
                Edit, Print, Screen, Lisp, Quit:
```

The system then waits for the user to select one of the options by entering the first letter of its name (**E, P, S, L** or **Q**).

Let's list the purposes of the main menu items:

- **Edit** - displays the prompt **Edit file:** and waits for the user to enter the name of the file to be edited. If you change your mind, just press **Enter** to exit to the main menu. Otherwise, enter the file name (for example, **C:EXPERT.LSP**) and press **Enter**. The default file type is **LSP**, as well as the registered device. The editor will read and display the file on the screen, "moving" the cursor to the upper left corner of the screen;

- **Print - displays the Print file:** prompt on the screen and waits for the user to enter the file name to print. If you change your mind, press **Enter** to exit to the main menu. Otherwise, enter the file name and press **Enter**, after making sure that the printing device is ready for work. The editor will start printing the file;
- **Screen** - displays **the Full, Vertical, Horizontal:** prompt on the screen and waits until the user selects one of the options (**F, V** or **H**):
  - **Full screen *(full screen)*** uses the full screen for editing;
  - ***Vertical* split screen** uses the right half of the screen for editing and the left half for debugging;
  - **Horizontal split screen *uses*** the top half of the screen for editing and the left half for debugging.

  If you need to change the screen, press the ***space bar*** ;

- **Lisp** - exits the editor; a dollar sign appears on the screen. When you have finished working in the **muLISP** environment, you can return to the editor by using the command **(RETURN)** ;
- **Quit** - terminates the editor and the **muLISP** system and transfers control to the operating system.

Keyboard commands.

The muLISP editor is a general-purpose screen-oriented editor; it consistently outputs a "window" or "picture" of text to the display screen. Therefore, the user can make any changes and see the results immediately.

Since the editor is written in the **muLISP** language, the user can supplement the system as needed. When a control character is entered, the corresponding function is called, performing the operation required by the user. Usually, this function can be redefined to perform any required task.

The general strategy for working with the **muLISP** editor can be expressed in two steps:

- move the cursor to the part of the text you want to edit;
- insert, delete or change text starting from the marked place.

The muLISP editor can be either **WordStar-like** or **Emasc-like**. To simplify the following descriptions, we will assume that the editor is **WordStar-like**.

Cursor control commands.

Cursor control commands are used to move the cursor to the point in the text where you want to make changes.

The cursor (the block of glowing lines indicating where the next character entered will appear) is controlled using control characters. For most consoles, control characters are entered by pressing the **Ctrl** key, and simultaneously the corresponding letter.

Cursor control is performed mainly according to the "+" sign rule, i.e. for standard keyboards, the position of the key corresponds to the direction in which the cursor moves.

This rule is illustrated by the following diagram (Fig. 1):

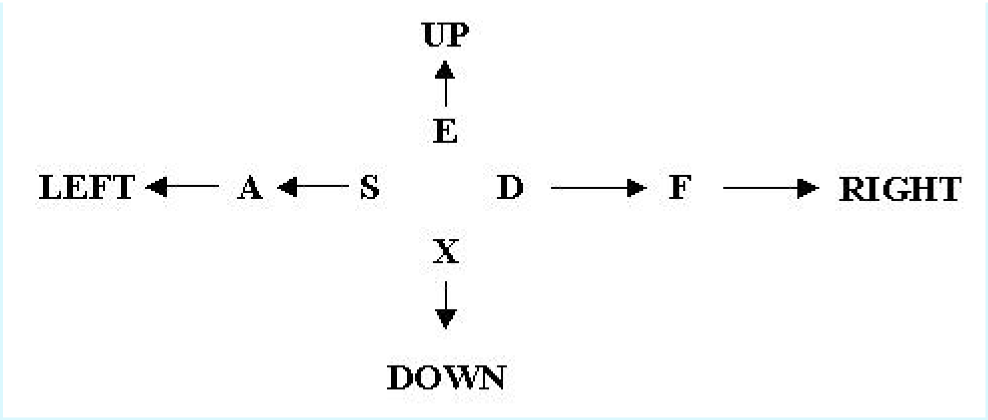


Fig. 1. Cursor control

The arrows indicate the direction in which the cursor will move when entering the corresponding control character. The length of the arrows indicates the relative magnitude of the cursor movement.

- **Ctrl-D** and **Ctrl-S** move the cursor to the right and left by a character, respectively.
- **Ctrl-H** or the **BACKSPACE** key also moves the cursor left one character.
- **Ctrl-F** and **Ctrl-A** move the cursor to the right and left of a word, respectively, shifting the cursor to the beginning of the word.

A word is a sequence of characters in text, delimited by a space, comma, colon, semicolon, beginning or end of a line.

Note that these four commands assume that the text on the screen is one long sequence of characters. Therefore, commands that normally move the cursor to the right automatically move it down to the beginning of the next line once the right end of the current line is reached. Commands that normally move the cursor to the left automatically move it up to the end of the previous line once the left end of the current line is reached.

- **Ctrl-E** and **Ctrl-X** move the cursor up and down a line, respectively.
- In Replace mode, Ctrl **-M** or the **TAB** key moves the cursor to the right to the next column whose number is a multiple of 8.
- **Ctrl-M** or the **ENTER** key moves the cursor down to the beginning of the next line.

If there is also a line of text below the cursor, **Ctrl-J** moves the cursor to the beginning of that line of text. Otherwise, these commands move the cursor down a line and automatically mark a paragraph according to the number of open and closed brackets and the cursor's original line.

Much less frequently used are cursor control commands that are initiated by entering a pair of characters. In general, pairs of commands that begin with **Ctrl-Q** enhance the action of the commands designated by the second character of the pair. Thus, **Ctrl-Q, D** moves the cursor to the right end of the current line, **Ctrl-Q, S** - to the left end of the current line, **Ctrl-Q, X** - to the bottom of the screen, **Ctrl-Q, E** - to the top of the screen.

Ctrl-Q, I moves the cursor left to the "previous" tab stop.

Ctrl-Q, P moves the cursor to the position it was in before movement commands such as **Ctrl-Q, X** or **Ctrl-Q, E** were executed.

Display screen control commands.

At any given moment, only part of the text can be shown on the display screen. If the editor receives a command to move the cursor to a part of the text that is not on the screen, the text "rolls up" either upwards (i.e. toward the end of the file) or downwards (toward the beginning of the file), and the text appears on the screen from the place to which the cursor is moved by command.

- **Ctrl-Z** moves the text up one line, and the new line will be located at the bottom of the screen.
- **Ctrl-W** moves the text down one line, and the new line will be located at the top of the screen. The cursor stays in the same position in the text, moving up or down with the text. However, if the cursor is at the top or bottom of the screen, it stays in place. The cursor never disappears from the screen!
- **Ctrl-R** and **Ctrl-C** move the text down and up, respectively, by 3/4 of the full screen. The cursor remains in the same position, but moves along with the text.
- **Ctrl-Q, C** moves the cursor and window to the end of the file, **Ctrl-Q, R** - to the beginning of the file.

Text input commands.

There are two modes for entering text: Replace mode **and** Insert **mode**.

- **Ctrl-V** is a switch from one input type to another. The name of the current input type is always indicated in the editor's status bar at the top right of the editor window. The editor is always initially set to the **Insert** type.

In **Replace** mode, characters and spaces under the cursor and to the right of it will be replaced by newly entered characters. This is the easiest way to initially enter text. Erroneous characters can be corrected by simply replacing them with correct ones from the keyboard, and text can be "erased" by replacing it with spaces.

In **Insert** mode, characters and spaces under and to the right of the cursor are shifted to the right, making room to the left for new characters inserted before them. For example, the text **(CONS BETA)** can be changed to **(CONS ALPHA BETA)** by inserting **ALPHA** before **BETA**.

- In **Insert mode, Ctrl-I** or the **TAB** key inserts a character into the text and moves the cursor to the right to the next tab stop.
- **Ctrl-M** or the **ENTER** key inserts a new line into the text and moves the cursor down to the beginning of a new line.
- In both **Insert** and **Replace modes, Ctrl-N** inserts a new line into the text; however, unlike **ENTER,** the cursor does not move, but remains at the right end of the broken line.
- **Ctrl-P** "forces" the editor to ignore the normal cursor control, and the next character you type will be added to the text like other characters.

Commands for destroying text.

In **Replace** mode, the easiest way to kill text is to replace it with spaces or other characters. Below are the kill commands that work for any type.

- **Ctrl-G** destroys the character under the cursor.
- **Ctrl-_** and **DELETE** delete the character immediately to the left of the cursor.
- **Ctrl-T** deletes the word to the right of the cursor.
- **Ctrl-Y** kills the line the cursor is on, including newline characters at the end of that line.
- **Ctrl-Q, Y** kills the right end of the current line, starting at the cursor and continuing to the end of the line, but not including newline characters.

In addition, there are block and list structure commands that are also used to destroy text.

Text that was last destroyed by the **Ctrl-Y** command or the block or structure destruction commands can be restored with **Ctrl-U**. This command allows you to restore text that was destroyed by mistake and also makes it easier to move or copy a line of text.

To move a line, move the cursor to the line and type **Ctrl-Y** ; the line will be deleted. If the line was previously copied, type **Ctrl-U**. Then move the cursor to the place where you want to paste the line and type **Ctrl-U**.

Block commands.

Block commands allow you to move, copy, delete a block of text, and also read it from a file or write it to a file. Two-character commands starting with **Ctrl-O** initiate block commands. The second character of the commands is mnemonic. A few seconds after entering **Ctrl-O**, a hint is displayed on the display screen if the user has not yet entered the second character of the command.

Similarly, for the **WordStar** editor type, two-character commands beginning with **Ctrl-K** can also initiate block commands. However, the second character of the commands is not always mnemonic.

- **Ctrl-O, B** and **Ctrl-K, B** mark the beginning of a block of text.
- **Ctrl-O, E** and **Ctrl-K, K** - end of text block.
- **Ctrl-O, M** and **Ctrl-K, V** move a block of text from its current position to the cursor position.
- **Ctrl-O, C** and **Ctrl-K, C** make a copy of the current block of text at the location indicated by the cursor.
- **Ctrl-O, D** and **Ctrl-K, Y** destroy the current block of text.
- **Ctrl-O, R** and **Ctrl-K, R** display the prompt: **Read file:** and wait until the user enters a file name and presses **ENTER** or **ESC**. If no file name is entered, the read command is aborted. Otherwise, the text is read from the file and inserted at the location indicated by the cursor.
- **Ctrl-O, W** and **Ctrl-K, W** prompt: **Write file:** and wait until the user enters a file name and presses **ENTER** or **ESC**. If no file name is entered, the write command is aborted. Otherwise, the text in the block is written to the file.

Find and replace commands.

- **Ctrl-Q, F** displays the prompt **Find string:** and waits for the user to enter a string and press **ENTER** or **ESC**. If no string has been entered, the command aborts. Otherwise, the command searches for the first equivalent string in the text. If the string is found, the cursor moves to the beginning of the string; if not, the cursor moves to the end of the text.
- **Ctrl-Q, A** displays the prompt: **Find string:** and waits for the user to type a string and press **ENTER** or **ESC**. Then the command displays the prompt: **Replace with:** and waits for the user to type a new line and press **ENTER** or **ESC**. Finally, the command searches for the first string in the text that matches the string from **Find string** and displays the prompt: **Replace string? (Y/N)** If the user types **"Y"**, the string is replaced with the string from **"Replace string"**. If the string from **"Find string"** is not found, the cursor moves to the end of the text.
- **Ctrl-L** repeats the search through the text for a string equivalent to the string in **"Find string"**.
- **Ctrl-Q, V** moves the cursor to the point where the last find or paste operation was performed.

List structure commands.

List structure commands allow you to move, copy, destroy, read from a file, and write to a file S-expressions.

Two-character commands beginning with **ESC** initiate list structure commands. If **muLISP** is running on **an IBM PC, the Alt** key can also be used. The second character of the **ESC** or **Alt** pair is a mnemonic.

- **ESC, D,** and **Alt-D** move the cursor forward one S-expression.
- **ESC, S** and **Alt-S** - back one S-expression.
- **ESC, F**, and **Alt-F** move the cursor forward, placing it after the end of the list.
- **ESC, A,** and **Alt-A** move the cursor back, placing it before the beginning of the list where the cursor is located.
- **ESC, X,** and **Alt-X** move the cursor forward and place it after the first left parenthesis (i.e. forward and down the list).
- **ESC, C,** and **Alt-C** move the cursor down to the beginning of a line whose first character is neither a space nor a semicolon (usually to the beginning of the next definition).
- **ESC, R,** and **Alt-R** move the cursor up to the beginning of a line whose first character is neither a space nor a semicolon.

- **ESC, T,** and **Alt-T** destroy the S-expression under the cursor and to the right of the cursor.
- **ESC, Y,** and **Alt-Y** destroy the definition where the cursor is located (i.e., move the cursor to the beginning of the current definition and destroy the S-expression).
- **ESC, L,** and **Alt-L** "close" the editor and transfer control to the **LISP** evaluation window. To return to the evaluation editor, type **(RETURN)** followed by the dollar sign.
- **ESC, !,** and **Alt-! print the result of evaluating the S-expression under and to the right of the cursor to the LISP** evaluation window. This command allows you to selectively redefine the definitions of the functions you are editing; this is useful when testing them.

File saving commands.

These commands allow you to save text or refuse to save it. Two-character commands starting with **Ctrl-K** initiate these commands.

A few seconds after entering **Ctrl-K**, a hint appears on the display screen if the user has not managed to enter the second character of the command.

- **Ctrl-K, D** saves the text as a file and returns control to the editor's main menu.
- **Ctrl-K, S** saves the text as a file and reopens the file for editing. If changes have been made to the text, **the Ctrl-K, A** and **Ctrl-K, Q** commands display a hint: **Abandon? (Y/N)**, where **filename** is the name of the edited file. If you enter **"Y"**, the edited text is deleted and control returns to the editor's main menu. Otherwise, no action occurs. If no changes were made, the text is simply deleted.

Window commands.

Ctrl-Q, W allows you to change the size of the text window and its shape while editing the file. The command displays a hint on the display screen:

```
              Full, Vertical, Horizontal
```

and waits until the user selects one of the options (**F, V** or **H**).

Theory of operations.

Since the editor is written in the **muLISP** language, a programmer with programming experience can change the existing editor functions or even add new ones. However, making complex changes, such as changing keyboard commands or adding new commands to the editor, requires the programmer to understand how the text saving process works and how the editor's text processing functions and text pointers can be used.

In the following, ***the pointer*** will be understood as the position in the text between the character on which the cursor is located and the character immediately to the left of it.

The text is saved by the editor as a list of elements, the P-names of which are sequences of characters in a text line.

- ***BELOW-POINT*** is the line of text below the pointer.
- ***ABOVE-POINT*** is the line of text above the pointer. The first line below and above the pointer is the first element of ***BELOW-POINT*** and ***ABOVE-POINT***, respectively. The second line is the second element, and so on.
- **If the *UNPACKED*** flag is NIL, then the first element of ***BELOW-POINT*** is the string containing the pointer. If ***UNPACKED*** is not **NIL**, then the first element of ***BELOW-POINT*** is the string below the string containing the pointer, and ***RIGHT-POINT*** and ***LEFT-POINT*** are the strings of characters to the right and left of the pointer, respectively. The first character to the right and left of the pointer is the first element of ***RIGHT-POINT*** and ***LEFT-POINT*,** respectively. The second character is the second element, and so on.

- ***POINT-ROW*** and ***POINT-COL*** are non-positive integers that contain the current position numbers of the pointer - row and column relative to the beginning of the text.
- ***CURSOR-ROW*** and ***CURSOR-COL*** contain the current position numbers of the cursor - row and column relative to the left corner of the current editor window.
- **The JUMP-POSN** function is used to move the pointer within text. (**JUMP-POSN *line column***) moves the pointer to the line of text with number ***line*** and the column of text with number ***column***.

Functions of editor commands.

Table 1 lists the editor command functions and their corresponding keyboard control characters in alphabetical order for both types of editors (**WordStar-like** and **Emacs-like**).

If you use the **muLISP** editor to develop your programs, do not give your **muLISP** program functions the names shown in this table.

Table 1. **Hot key combinations**

Function	Type WordStar	Type Emasc
BOTTOM-ROW	Ctrl-X	
COPY-BLOCK	Ctrl-O C	Ctrl-U C
DEL-BLOCK	Ctrl-O D	Ctrl-U D
DEL-DEFN	Esc Y	
DEL-LEFT-CHAR	RUBOUT	Ctrl-H
DEL-LEFT-END	Ctrl-Q RUB	Ctrl-X K
DEL-LEFT-WORD	Ctrl-Q T	RUBOUT
DEL-LINE	Ctrl-Y	
DEL-RIGHT-CHAR	Ctrl-G	Ctrl-D
DEL-RIGHT-END	Ctrl-Q Y	
DEL-RIGHT-LINE		Ctrl-K
DEL-RIGHT-WORD	Ctrl-T	Ctrl-X D
DEL-SEXP	Esc T	Ctrl-Z K
DOWN-LINE	Ctrl-X	Ctrl-N
DOWN-LINE-INDENT	Ctrl-J	Ctrl-J
END-TEXT	Ctrl-Q C	Ctrl-X >
ESCAPE-CHAR	Ctrl-P	Ctrl-Q
EVAL-LISP	Esc L	Ctrl-Z !
EVAL-SEXP	Esc !	Ctrl-Z E
FIND-NEXT	Ctrl-L	Ctrl-L
FIND-CTRL	Ctrl-Q F	Ctrl-S
INSERT-LINE	Ctrl-N	Ctrl-O
INSERT-MODE	Ctrl-V	Ctrl-X I
JUMP-END	Ctrl-Q K	
JUMP-FIND	Ctrl-Q V	
JUMP-LAST	Ctrl-Q P	
JUMP-START	Ctrl-Q B	
LEFT-CHAR	Ctrl-S	Ctrl-B

LEFT-END	Ctrl-Q S	Ctrl-A
LEFT-LIST	Esc A	Ctrl-Z P
LEFT-SEXP	Esc S	Ctrl-Z B
LEFT-TAB	Ctrl-Q I	
LEFT-UP-LIST	Esc E	Ctrl-Z (
LEFT-WORD	Ctrl-A	Ctrl-X B
MARK-END	Ctrl-O E	Ctrl-U E
MARK-START	Ctrl-O B	Ctrl-U B
MOVE-BLOCK	Ctrl-O M	Ctrl-U M
NEXT-DEFN	Esc C	Ctrl-Z]
NEW-LINE	Ctrl-M	Ctrl-M
QUIT-EDIT	Ctrl-K A	Ctrl-X A
READ-BLOCK	Ctrl-O R	Ctrl-X R
REPL-STRG	Ctrl-Q A	Ctrl-X %
RIGHT-CHAR	Ctrl-D	Ctrl-F
RIGHT-DOWN-LIST	Esc X	Ctrl-Z D
RIGHT-END	Ctrl-Q D	Ctrl-E
RIGHT-LIST	Esc F	Ctrl-Z N
RIGHT-SEXP	Esc D	Ctrl-Z F
RIGHT-TAB	Ctrl-I	Ctrl-I
RIGHT-UP-LIST	Esc Z	Ctrl-Z)
RIGHT-WORD	Ctrl-F	Ctrl-X F
SAVE-EDIT	Ctrl-K S	Ctrl-X S
SAVE-FILE	Ctrl-K D	
SCROLL-DOWN-LINE	Ctrl-W	
SCROLL-DOWN-SCRN	Ctrl-R	Ctrl-X V
SCROLL-UP-LINE	Ctrl-Z	
SCROLL-UP-SCRN	Ctrl-C	Ctrl-V
SPLIT-WINDOW	Ctrl-Q W	Ctrl-X 2
START-TEXT	Ctrl-Q R	Ctrl-X <
THIS-DEFN	Esc R	Ctrl-Z [
TOP-ROW	Ctrl-Q E	
UNDELETE	Ctrl-U	Ctrl-Y
UP-LINE	Ctrl-E	Ctrl-P
WRITE-BLOCK	Ctrl-O W	Ctrl-X W

Global editor variables.

Table 2 lists the editor's global variables in alphabetical order and briefly defines their values.

If you use the **muLISP** editor to develop your programs, do not give variables or constants in your programs the names shown in the table.

Table 2. **Global editor variables**

Variable	**Definition**
ABOVE-POINT	Text above the pointer.
BASE-COL	The number of the source column of the current window.
BASE-ROW	The source line number of the current window.
BELOW-POINT	Text below the pointer.
COLUMNS	The number of columns in the current window.
CURSOR-COL	The position of the cursor in the column in the current window.
CURSOR-ROW	The position of the cursor in a line in the current window.
DELETED-TEXT	The last text destroyed.
END-COL	The number of the last column of the text block.
END-ROW	The number of the last line of a block of text.
EVAL-ROW	The current line number of the **LISP** evaluation window.
FILE-NAME	The name of the last edited file.
FIND-COL	The result string of the last search/replacement.
FIND-ROW	Column - result of the last search/replacement.
FIND-CTRL	Current value of **"Find string"**.
INSERT	**T** if type is **Insert, NIL** if type **is Replace**.
LAST-COL	Column - the result of the last move.
LAST-ROW	The result string of the last move.
LEFT-POINT	List of symbols to the left of the pointer.
POINT-COL	Column number of the index in the text.
POINT-ROW	The line number of the index in the text.
REPL-CTRL	Current value of **"Replace string"**.
RIGHT-POINT	List of symbols to the right of the pointer.
ROWS	The number of lines in the current editor window.
SCREEN	List of R-names displayed on the display.
START-COL	The number of the first column of the text block.
START-ROW	The number of the first line of a block of text.
STAT-COL	Column number in the status window.
STAT-INS	Not **NIL** if and only if the current type is returned.
STAT-ROW	Line number in the status window.
STAT-WINDOW	Not **NIL** if and only if the file name is returned.
TEXT-DIRTY	Not **NIL** if and only if the text has changed.
UNPACKED	Not **NIL** if and only if the text string is unpacked.

Global editor constants.

Table 3 lists the editor's global constants, their default values, and brief descriptions of their usage in alphabetical order.

If you use an editor to develop programs, do not give variables or constants in your programs the names shown in this table.

Table 3. **List of constants**

A constant	**Meaning**	**Description**
ATOM-DELIMITER	**(\\(\\) | | <tab>)**	Symbols that delimit atoms.
BLANK	**| |**	Empty symbol.
BLINK-PAPEN	**T**	Flickering image parameters.
DEFAULT-TYPE	**LSP**	Default file extension.
LPAR	**\\(**	Left brackets.
NUL-CTRL	**""**	Empty line.
PAGE-LINES	**66**	Printed page line.
RPAR	**\\)**	Right brackets.
TAB	**<tab>**	Tab character.
WHITESPACE	**(| | <tab>)**	Whitespace characters.
WORD-DELIMITER	**(\\, \\: \\; | | <tab>)**	Symbols that delimit words.

In the next step we will look at the rules for working with the display screen in **muLISP-85**.

Step 32.
Working with the Display Screen in muLISP-85

In this step we will look at ways to control the screen in the **muLISP-85** system.

Display screen functions are used to display and format text on the screen - these are functions for creating windows, changing the cursor position, and "folding" text. Display screen variables provide control over attributes such as flickering, inverted image, brightness, and automatic folding.

The procedures that **muLISP** can use to perform various screen operations vary from computer to computer and/or display to display. Therefore, when **muLISP first starts up, it tries to detect the type of computer immediately. If it succeeds, the second line of the muLISP** message indicates this.

If the system cannot determine the type of computer, another message is displayed:

```
             1 = Other generic MS-DOS computer
             2 = IBM PC or "look-alike" computers
             3 = ANSI.SYS screen or VT-100 Terminal
             4 = TI Professional Computer
             5 = Zenith Z-100 Computer or VT-52 Terminal
             6 = Hewlett-Packard HP-150 Computer
             7 = Hewlett-Packard HP-110 Computer
             8 = NEC Advanced Personal Computer or ADM-3A Terminal

                   Please enter your computer type number:
```

If your computer type is not specified here, then select type “1”. However, in this case, many of the actions considered in this application will be invalid and will return **NIL**.

A muLISP program can determine the type of computer by examining the type byte, which is located at offset 855 decimal in the **muLISP base page.**

Therefore, the function:

```
        (CSMEMORY 855)
```

will return the computer type number - a positive integer.

Further we will assume that the display screen has ***25 rows and 80 columns***. For screens of other sizes, these values should be replaced with the required ones, according to the type of screen.

The upper left corner of the screen (home position) corresponds to row zero and column zero. ***The display screen cursor*** is a flickering beam of light on the screen, consisting of one or more horizontal lines of 1 character each. For a monochrome display, the cursor may include up to 14 lines, for a graphic display - up to 8 lines. In any case, the cursor lines are numbered starting from 0.

Let us list the functions and control variables of the **muLISP-85** system, intended for working with the display screen.

Table 1. **Functions controlling the screen operation**

Function	**Purpose**
(SET-CURSOR ROW COLUMN)	Move the cursor to the desired row and column on the screen.
(ROW) or (COLUMN)	Return the position of the cursor in the row and column according to the current window.
(CLEAR-SCREEN)	"Cleans" the current window.
(INSERT-LINES N)	Inserting empty lines.
(DELETE-LINES N)	Deleting lines.
(MAKE-WINDOW ROW COLUMN ROWS COLUMNS)	Creating a window.
(FOREGROUND-COLOR N) or (BACKGROUND-COLOR N)	Set text and background color.
(CURSOR-LINE START-LINE END-LINE)	Sets the cursor start and end lines.
(DISPLAY-PAGE N)	Setting up a video memory page.
AUTO NEWLINE	Enable/disable automatic insertion of new lines.
BLINK	Enable/disable blinking of symbols.
HIGH-INTENSITY	Enable/disable high brightness.

1. If **ROW** and **COLUMN** are 0 or positive integers less than 25 and 80 respectively, then the function **(SET-CURSOR ROW COLUMN)** moves the cursor to the desired row and column of the screen and returns **T**. Otherwise, **NIL** is returned. The example shows how to return the cursor to its original position; in this case, "T" is displayed in the upper left corner of the window:

```
            $ (SET-CURSOR 0 0)
            T
```

The (SET-CURSOR) function returns a list of two numbers that specify the current cursor position.

2. The functions **(ROW)** and **(COLUMN)** return the position of the cursor in the row and column according to the current window.

3. The **(CLEAR-SCREEN)** function "clears" the current window, moves the cursor to the upper left corner of the window (to its original position), and returns **T**. Otherwise, **NIL** is returned. The example shows how to clear the screen; in this case, **T** is returned to the upper left corner:

```
            $ (CLEAR-SCREEN)
            T
```

4. If **N** is zero or a positive integer, then the function **(INSERT-LINES N)** inserts **N** "clean" lines into the current window, starting from the line marked by the cursor, and returns **T**. Otherwise, **NIL** is returned.

New lines are inserted by "collapsing" the required number of lines in the lower part of the window.

5. If **N** is zero or a positive integer, then the function:

```
            (DELETE-LINES N)
```

destroys **N** lines of the current window, starting from the line marked by the cursor, and returns **T**. Otherwise, **NIL** is returned.

The lines are destroyed by "collapsing" the required number of lines below the cursor upwards (empty lines are added to the bottom of the screen).

6. If **ROW, COLUMN, ROWS** and **COLUMNS** are within certain numeric limits, then the function:

```
            (MAKE-WINDOW ROW COLUMN ROWS COLUMNS)
```

creates a rectangular region on the console screen as the current window, moves the cursor to the upper-left corner of the window, and returns **T. The upper-left corner of the window is determined by the values of the ROW** and **COLUMN** arguments. The **ROW** argument must be zero or a positive integer less than 25. **The COLUMN** argument must be zero or a positive integer less than 80. Both **ROW** and **COLUMN** default to 0.

The window has a width of **COLUMNS** and a height of **ROWS**.

The ROWS argument must be a positive integer less than or equal to **(25 - ROW)**. **ROWS** is treated as **(25 - ROW)** by default.

The COLUMNS argument must be a positive integer less than or equal to **(80 - COLUMN)**. **COLUMNS** is treated as **(80 - COLUMN)** by default.

The (MAKE-WINDOW) function returns a list of four elements: the source row, the source column, the number of rows, and the number of columns of the current window.

The example shows how to create a window with 12 rows and 40 columns in the center of the screen:

```
            $ (MAKE-WINDOW 6 20 12 40)
            T
```

7. If **N** is zero or a positive integer less than 16, the functions:

```
            (FOREGROUND-COLOR N) or
            (BACKGROUND-COLOR N)
```

set the foreground (text) and background (border) background to **N,** respectively, and return the previous value. For example, the function:

```
            $ (FOREGROUND-COLOR 2)
             7 ; returned the gray code
```

will set the text color to green.

If functions are called without arguments, they return the code of the current text color and border color, respectively. For example, the **(FOREGROUND-COLOR)** function returns the code of the current text color.

If **muLISP** is running on **an IBM PC** with a monochrome monitor, the following results are displayed on the display screen for various values of **N** (Table 2):

Table 2. **Setting color on monochrome monitors**

Foreground	**Background**	**Result**
0	**0**	No image.
1	**0**	Underline.
7	**0**	Normal image.
0	**7**	Inverted image.

If **muLISP** is running on **an IBM PC** with a color graphics display, then the different values of **N** correspond to colors (Table 3):

Table 3. **Color assignment for color monitors**

Color number	**Color**
0	Black.
1	Blue.
2	Green.
4	Red.
5	Magenta.
6	Brown.
7	Light grey.
8	Dark grey.
9	Light blue.
10	.Light green
11	Light cyan.
12	Light red.
13	Light fuchsia.
14	Yellow.
15	White.

8. If **muLISP** is running on an **IBM PC** and if **START-LINE** and **END-LINE** are non-negative integers less than the maximum number of cursor lines for the monitor, then the function **(CURSOR-LINE START-LINE END-LINE)** sets the cursor start and end lines and returns a list consisting of the numbers of the "old" start and end lines. Examples:

```
$ (CURSOR-LINES 0 0)
 ; the cursor takes the form of a horizontal
 ; features at the top of the character space
$ (CURSOR-LINES 1 0)
 ; invisible cursor
$ (CURSOR-LINES 0 13)
(12 13) ; Cursor as a filled block
```

The (CURSOR-LINES) function returns a list **(N1 N2) of two numbers specifying the shape of the cursor. When muLISP** finishes, it restores the original cursor shape.

9. If **muLISP** is running on an **IBM PC** with a graphics display, and if **N** is zero or a positive integer less than

8, then the function **(DISPLAY-PAGE N)** sets the current video memory page to page **N** and returns the number of the previous video page.

Otherwise, the **DISPLAY-PAGE** function returns the current display page number. This allows you to quickly alternate between the contents of several text-filled display screens.

10. If the control variable ***AUTO-NEWLINE*** is not **NIL**, then immediately after the symbol that is displayed in the rightmost column of the current window, a new line automatically appears. If ***AUTO-NEWLINE* is NIL**, then a new line does not appear, and the cursor is removed from the rightmost column of the window. **muLISP-90**: if this variable is not **NIL**, all **WRITE** functions to a file also inserts CR/LF pairs into the output stream after every 79 characters.

11. If the control variable ***BLINK*** is not **NIL**, then new characters on the display screen will blink, if **NIL**, they will not blink. Examples:

```
             $ (SETQ *BLINK* T)
             ; flashing symbols
             $ (SETQ *BLINK* NIL)
              ; "regular" characters
```

12. If ***HIGH-INTENSITY*** is not **NIL**, then new symbols will be displayed with increased brightness; if **NIL** - with decreased brightness (dimly). Examples:

```
              $ (SETQ *HIGH-INTENSITY* T)
              ; increased brightness
              $ (SETQ *HIGH-INTENSITY* NIL)
              ; normal brightness
```

In the next step we will list the error messages in the **muLISP-85** system.

Step 33. Error messages in the muLISP-85 system

In this step we will list the error messages output by the **muLISP-85** system.

When **muLISP** detects an error condition, the **BREAK** function is called. **BREAK** issues an appropriate error message, suspends program execution, and provides the user with options for continuing execution:

```
             Continue, Break, Abort, Top-level, Restart, System?
```

The system then waits for the user to select one of the options by specifying its name (**C, B, A, T, R** or **S**, respectively). Note that the options are listed in order of increasing effectiveness.

- **Continue - returns control to the program that caused the interrupt (stop). If the cause of *the*** interrupt was an interrupt command sent from the keyboard, execution continues as if the interrupt (stop) had not occurred. If the interrupt occurred as a result of an error delay, the value passed at the interrupt by the error regulator is returned as the value of the error function;
- **Break -** temporarily suspends program execution and exits to the next lower level of the read-eval-print loop. **This** allows the user to examine and/or change the current **muLISP** environment before continuing program execution. To exit a break and resume program execution, type **(RETURN)** followed by a dollar sign;

- **Abort (*interrupt*)** - interrupts program execution, assigns initial values to formal parameters placed in the variable stack, and returns control to the current level of the **"read-eval-print"** cycle. Function definitions, property values, and global variables remain unchanged;
- **Top-level (*upper level*)** - interrupts program execution, assigns initial values to formal parameters, which are placed in the variable stack, displays current input and output data on the console and returns control to the upper level of the **"read-eval-print"** cycle. Function definitions, property values and global variables remain unchanged;
- **Restart -** closes all open files, discards the current muLISP environment, **and** initiates a new **muLISP** system. All variable associations, functions, and property values in the current **muLISP** environment are destroyed;
- **System (*system*)** - closes all open files, terminates **muLISP** execution and returns control to the operating system.

Below are the error messages in the **muLISP-85** system, listed in alphabetical order:

- **DISK FULL - means** that there is not enough memory to accommodate the data written to the disk. The program execution stops and an error interrupt occurs. Since the file remains open, it is possible to delete other files on the entire diskette (using the **EXECUTE** function) and continue writing to the file;
- **END-OF-FILE (*end of file*)** - means that an attempt was made to read data beyond the end of the input file or from its empty spaces (see the description of the **WRITEPTR** function). Following the message **"end-of-file"** a list of the type **"drive:name.type"** is displayed ;
- **FILE NOT FOUND -** means that the source and/or **SYS** file specified in the **OS** commands initiating **muLISP** was not found, or **the SYS** file is of the "wrong" version. **A SYS** file can only be loaded under the version of **muLISP** that is used to save the file. Source and **SYS** files can also be loaded into **muLISP** using the **RDS** and **LOAD** commands, respectively. When one of these commands completes and the file is not found, the command returns the **NIL flag instead of the "FILE NOT FOUND"** message ;
- **INSUFFICIENT ARGUMENTS -** means that a function that requires at least one argument is called without arguments. Functions that can cause this type of error are: **MAX, MIN, -, /, ADD1, SUB1, LCM, ABS, SIGNUM, NUMERATOR, DENOMINATOR, FLOOR, CEILING, TRUNCATE, ROUND, MJD, REM, DIVIDE, LOGNOT, BITLENGTH,** and **SHIFT** ;
- **INSUFFICIENT MEMORY, ABORTING - means** that there is not enough memory to load and run the **muLISP** environment. **muLISP** is suspended and control is returned to the host **OS**. Note that the **muLISP** environment, stored in a **SYS** file, can be loaded into a computer that has less memory than the computer on which it was created. An out-of-memory error occurs only when the computer on which **the SYS** file was loaded does not have enough memory to accommodate the **muLISP** environment. The only way to load **SYS** files is to obtain more memory for the computer.
- **MEMORY FULL -** means that there is not enough memory to continue executing **muLISP** programs. Program execution is suspended and an error interrupt occurs. Indeed, the memory management system provides each data area with enough memory to fully satisfy the needs of muLISP programs. **If** the memory requirements for data objects exceed all available resources, this error occurs. Statistics in the following form are displayed along with the error message:

```
      GC: nnnn aaaa/aaaa vvvv/vvvv pppp/pppp ssss/ssss tttt/tttt
```

The hexadecimal digits following **"GC:"** indicate the amount of memory remaining in each of the main 4 data areas. Therefore, the data area associated with the error can be determined;

- ***NONINTEGER* ARGUMENT -** means that a function that requires integer arguments was called with a non-integer argument. Functions for which this error can occur are: **LOGAND, LOGIOR, LOGXOR, LOGNOT, SHIFT** and **BITLENGTH** ;
- **NONINTEGER ARGUMENT - means** that a function that requires numeric arguments was called with a non-numeric argument. This error can occur for the following functions: **=, /=, <, >, <=, >=, MAX, MIN, +, -, *, /, ADD1, SUB1, INCQ, DECQ, GCD, LCM, ABC, SIGNUM, NUMERATOR, DENOMINATOR, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM,** and **DIVIDE** ;

- ***NONSYMBOLIC* ARGUMENT -** means that a function that requires symbolic arguments is called with a nonsymbolic argument. Such functions include: **SET, SETQ, PSETQ, POP, PUSH, INCQ,** and **DECQ** ;
- **SYNTAX ERROR (*syntax error*)** - means that the **READ** function encountered either extra right parentheses or an inaccuracy in the bitmap, such as **(A .)** or **(A B .C D)**. Since the interrupt for this error is generated by the right parentheses or commas macro, it can be modified by the user-designer;
- **UNDEFINED FUNCTION - means** that an attempt was made to use a symbol that does not have a function definition. The general actions for this error are to select the **BREAK** option, define the undefined symbol, and continue the original program with the **(RETURN (EVAL BREAK))** command ;
- **ZERO DIVIDE (*division by 0*)** - means that a division function with a zero divisor was called. Such functions can be: **/, FLOOR, CEILING, TRUNCATE, ROUND, MOD, REM** and **DIVIDE**.

In the next step we will begin to study ***recursion and its forms***.

Step 34.
Recursion. General information

In this step we will introduce the concept ***of recursion***.

In the "pure" functional language **LISP** there are no cyclic statements (**LOOP**), and especially no control transfer operators. For programming repetitive calculations, it uses only **COND** functions and definitions of ***recursive functions***.

In [1, p. 66-67] one can read the following informal explanation of the essence of the recursion process: ***by recursion we mean such an organization of a complex system in which: a certain set of basic subsystems is distinguished; the system is capable of creating an unlimited number of copies of the basic systems during its operation, interacting with them and destroying them; the functioning of the system consists of the functioning of the basic subsystems and their active copies; when a copy is called, its change is permissible, determined by the situational environment at the time of the call.***

We will understand a ***recursive*** function as a function whose body contains a call to itself.

Iterative and recursive programs are theoretically identical in their computational capabilities, in other words, the sets of functions that can be calculated with their help coincide. So, in principle, any calculation can be programmed using any of these methods. However, the properties of iterative and recursive program variants can differ significantly. In this regard, it is often necessary to decide which of the programming methods is more suitable for a given task. The simplicity and naturalness of programming, as well as its efficiency in terms of execution time and memory use, depend on the choice made.

With iterative programming, which is usually longer and more difficult to implement, the result can be calculated much more simply and quickly. This happens for two reasons:

- Firstly, because von Neumann computers are generally oriented towards sequential computations, and,
- secondly, because translators are not always able to transform a recursive definition into an iterative one and use a stack in calculations, despite the fact that it is not always needed.

Recursive programming is generally shorter and more meaningful. It is especially useful to use recursion when the problem being solved or the data being processed is recursive in nature.

Recursion is best illustrated by working with lists, since lists themselves have ***a recursive structure***.

Indeed, recall that earlier (step 4. List as a fundamental data type) we defined a list using the following Backus-Naur rules: ***a list** is either empty or it is a dotted pair whose tail is **a list**.* There is recursion in the definition!

Recursive functions can be successfully used when working with other dynamic data structures (for example, trees) that have a recursive organization.

Recursion reflects an essential feature of abstract thinking, which manifests itself in the analysis of complex algorithmic and structural constructions in a wide variety of applications. The fact is that, using recursion, we get rid of the need for a tedious sequential description of the construction and limit ourselves to identifying only the relationships between the various levels of this construction. Recursion in the **LISP** language is not only the organization of calculations - it is a way of thinking and a methodology for solving problems.

---

(1) Anisimov A.V. Computer science. Creativity. Recursion. - Kiev: Nauk.dumka, 1988. - 224 p.

In the next step we will look at how to create recursive definitions.

Step 35.
How to write recursive definitions

In this step we will look at the rules for writing recursive definitions.

There are ***three*** different styles of recursive definitions; we will call them

- ***ascending***,
- ***descending recursion*** and
- ***tree recursion*** [1, p.72-73].

Descending recursion successively breaks down a given problem into increasingly simpler ones until it reaches ***a terminal situation***.

By ***the terminal case*** we mean a situation where no continuation of recursion is required. In this case, the value of the defined function is obtained without using an invocation of it (in relation to other values of the argument), which is typical for recursive definitions.

Only after this does it begin to build a response, and the intermediate results are passed back to the calling functions.

Here is a variant of calculating the factorial function using this technique:

```
(DEFUN FACT1 (LAMBDA (N)
   (COND ( (EQ N 1) 1 )
         (  T  (TIMES N (FACT1 (DIFFERENCE N 1))) ))))
```

Another example of downward recursion is given by the function for copying a list:

```
(DEFUN DCOPY (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         (  T  (CONS (CAR LST) (DCOPY (CDR LST))) ))))
```

In ***bottom-up recursion,*** intermediate results are calculated at each stage of the recursion, so that the answer is built up gradually and passed as a working memory parameter until a terminal situation is reached. By that time, the answer is ready and only needs to be passed to the upper-level caller.

We can write the bottom-up version of the **FAC** function as follows:

```
(DEFUN FAC (LAMBDA (N)
      (FACT N 1)
   ))
   ; ---------------------
   (DEFUN FACT (LAMBDA (N W))
      (COND ( (EQ N 0) W )
            (  T  (FACT (- N 1) (* N W)) ))))
```

Here **W** is the working memory parameter used to generate the results. Since the original function has only one parameter, it is necessary to define an auxiliary function **FACT** with an additional parameter, and the first time the **FACT** function is called, this parameter must be initialized (given an initial value).

The calculation can be done by substitution:

```
   FAC (3) = FACT (3,1) = FACT (2,3*1) = FACT (1,2*3*1) =
           = FACT (0,1*2*3*1) =1*2*3*1 = 6
```

We see that the result is formed here in the working memory parameter, and we notice that its value is known at each step, even though we write it as an expression like **2*3*1**. In contrast, the top-down version **of FAC** produces intermediate results like **3*2*FACT1(1)**, and we cannot compute this value until we know **FACT1(1)**.

Here is another example of ascending recursion for "reversing" a list:

```
   (DEFUN UCOPY (LAMBDA (LST)
      (COP LST NIL)))
   ; ----------------------
   (DEFUN COP (LAMBDA (LST W)
      (COND ( (NULL LST) W )
            (  T  (COP (CDR LST) (CONS (CAR LST) W)) ))))
```

Tree recursion (parallel recursion) is another type of recursion that has to be used even in "regular" assignment programming. It is used to traverse a tree structure, i.e. a list whose elements may themselves be lists. You will learn about parallel recursion and examples of its use in the corresponding section.

The process of writing a recursive function definition breaks down into discrete steps. These definitions seem obvious afterwards, but they are not always easy to come up with. The difficulty seems to be that we must repeatedly call the function being defined at a time when we do not yet know whether the function definition we are writing will work.

Here we need to assume that we already have a function that works correctly for an argument equal to N-1 or (CDR LST) and construct a definition that will work for the next case (i.e. for a number N or a list LST).

The recipes (!) for writing function definitions in these three cases are as follows [1, pp. 78-80].

1. ***Downward recursion (tail recursion)***.
   - Write a definition for a terminal case, such as the number 0 or the empty list **NIL**.
   - Now suppose you have a function definition that works for the case closest to the terminal (i.e., **N-1**, or the tail of the list). Try (!) to apply it to construct an expression that works for the next case as you go up.
   - Combine the results of steps 1 and 2 into a conditional expression. Test it thoroughly with examples.
2. ***Ascending recursion***.
   - Invent (!) a function that has one or more parameters of the working memory type, in one of which the result will be built.
   - For the terminal case, let the function value be equal to the working memory type parameter.
   - For the non-terminal case, re-call the function with the new parameter values expressed in terms of the old ones.

- Call the function with initial values of the working memory type parameters.
- Test the function definition on examples and "tweak" the initial values if necessary.

3. ***Tree recursion***.
   - Apply the predicate **(ATOM LST)** and, if necessary, **(NULL LST)** to recognize the terminal case.
   - In general, assume that your function works correctly for **(CAR LST)** and **(CDR LST)**, and write an expression for the combination of these values.

Due to the prescription nature of our advice, we provide a large number of examples in the sections devoted to recursion. It is advisable that the reader execute the written programs on a computer and try to improve them. This is one of the most famous ways to learn recursive programming!

---

(1) Gray P. Logic, Algebra and Databases / Translated from English. - Moscow: Mashinostroenie, 1989. - 368 p.

From the next step we begin to look at examples of using recursion.

Step 36.
Simple recursion. Recursion on arguments

In this step we will look at examples of using recursion on arguments.

We will say that recursion ***is simple*** if a function call occurs in some program branch only ***once***. Simple recursion in imperative programming corresponds to an ordinary cycle.

If a function returns the value of another function as a result, and the recursive call is involved in calculating the arguments of this function, then we will talk about the presence of ***recursion on arguments*** in the function.

Example 1. Calculating the factorial of a natural number **X.** The problem is solved using a function that takes a value equal to one when the argument is zero, and in all other cases is equal to the product of the argument and the value of the same function of the argument, reduced by one.

```
(DEFUN FACT (LAMBDA (X)
   (COND ( (ZEROP X) 1 )
         (  T  (* X (FACT (- X 1))) )
   )
))
```

Example 2. The **COPY** function, which builds a copy of the "topmost" level of the list **LST**:

```
(DEFUN COPY (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         (   T   (CONS (CAR LST) (COPY (CDR LST))))
   )
))
```

The constructed **COPY** function is recursive in its argument, since the recursive call is the argument of the **CONS** function. The **COND** function in the body of the function contains two branches: a branch with the termination condition and a branch with recursion, with the help of which the function traverses the list, copying and shortening the list in the process in the **CDR** direction or, as they say, ***in the width***.

Copying lists is one of the most basic operations on lists, and so the corresponding function is included among the built-in functions in almost all LISP systems.

Example 3. Define a function that removes the last element from a list.

```
(DEFUN DROPLAST (LAMBDA (LST)
   (REVERSE (CDR (REVERSE LST)))
))
```

Example 4. A function that removes the first occurrence of a given element at the top level from a list.

```
(DEFUN REMOVEF (LAMBDA (X LST)
   (COND ( (NULL LST) NIL )
         ( (EQ X (CAR LST)) (CDR LST) )
         (   T    (CONS (CAR LST) (REMOVEF X (CDR LST))) )
   )
))
```

Test example:

```
$ (REMOVEF (A B) (A B (A B) C (A B)))
(A B C (A B))
```

Example 5. A function that removes every second element from a list.

```
(DEFUN EVERYSECOND (LAMBDA (L)
   (COND ( (NULL L) NIL )
         ( (NULL (CDR L)) L )
         (  T  (CONS (CAR L) (EVERYSECOND (CDDR L))) )
   )
))
```

Example 6. Determining the largest element of a single-level numeric list **LST**.

```
(DEFUN MAX (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (EQ (LENGTH LST) 1)  (CAR LST) )
         (   T  (MAX2 (CAR LST) (MAXIM (CDR LST))) )
   )
))
; ---------------------
(DEFUN MAX2 (LAMBDA (X Y)
   (( (> X Y) X )
              AND )
))
```

What corrections need to be made in the program to obtain the smallest element of a single-level list?

Example 7. A function that, by alternating the elements of the lists **(A B ...)** and **(1 2 ...)**, forms a new list **(A 1 B 2 ...)**.

```
(DEFUN CHEREDUJ (LAMBDA (X Y)
   (COND ( (NULL X) Y )
         (   T    (CONS (CAR X) (CHEREDUJ Y (CDR X))) )
)))
```

Example 8. Counting the number of occurrences of atom **X** in the list **LST**.

```
(DEFUN NINSERT (LAMBDA (X LST)
   (COND ( (NULL LST) 0)
         ( (EQ  X (CAR LST))
                   (+ 1 (NINSERT X (CDR LST)) ) )
         (   T   (NINSERT  X (CDR LST)) )
   )))
```

Example 9. Finding the sum of the elements of a single-level numeric list.

```
   (DEFUN SUMMA (LAMBDA (LST)
      (COND ( (NULL LST) 0)
            (   T   (+ (CAR LST) (SUMMA (CDR LST))) )
      )
   ))
```

Example 10. Counting the number of atoms in a given list **LST** that are at the first level.

```
   (DEFUN  LENGTH_1 (LAMBDA (LST)
      (COND ( (NULL LST) 0 )
            (   T   (+ 1 (LENGTH_1 (CDR LST))) )
      )
   ))
```

Note that if you apply this function to a list of lists, the **NIL** elements that enclose each list will also be counted!

Example 11. A function that inserts an element **X** into **the N-th** position of the list **LST**:

```
   (DEFUN INSERT (LAMBDA (X N LST)
      (COND ( (NULL LST) (CONS X LST) )
            ( (EQ N 1) (CONS X LST) )
            (  T  (CONS (CAR (LST))
                        (INSERT X (DIFFERENCE N 1)
                                (CDR LST)))
            )
      )
   ))
```

Example 12. A function that transforms a list **(A B C D ...)** into a list of the form **((A B) (C D) ...)**.

```
   (DEFUN DOUBLE (LAMBDA (L)
      (COND ( (NULL L) NIL )
            ( (NULL (CDR L)) NIL )
            (  T  (CONS (LIST (CAR L) (CADR L))
                        (DOUBLE (CDDR L))) )
      )
   ))
```

Example 13. From two given lists **X** and **Y** of the same length, we construct a new list, the structure of which we will consider using an example: if **X** is a list **(A B C)**, and **Y** is a list **(1 2 3)**, then the new list should have the form **(A 1 (B 2 (C 3)))**.

```
   (DEFUN SASHA (LAMBDA (X Y)
      (COND  ( (NULL X) )
             (   T   (LIST (CAR X) (CAR Y)
                           (SASHA (CDR X) (CDR Y)))
             )
      )
   ))
```

Example 14. From two given lists **X** and **Y** of the same length, we construct a new list, the structure of which we will consider using an example: if **X** is a list **(A B C)**, and **Y** is a list **(1 2 3)**, then the new list should have the form **((A 1) (B 2) (C 3))**.

```
(DEFUN UNIT (LAMBDA (X Y)
   (COND ( (NULL X) NIL)
         ( (EQ (LENGTH X) 1)
                     (LIST (CAR X) (CAR Y)) )
         (   T   (LIST (LIST (CAR X) (CAR Y))
                       (UNIT (CDR X) (CDR Y))) )
   )
))
```

Example 15. Rearrange the elements of a given list so that identical elements, if they are in the list, are arranged in a row.

```
(DEFUN COLLECT (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         (  T  (CONS (CAR LST)
                     (COLLECT (COND ( (MEMBER (CAR LST)
                                             (CDR LST))
                                     (CONS (CAR LST)
                                           (REMOVEF
                                              (CAR LST)
                                              (CDR LST))))
                                   (  T  (CDR LST))))) )
   )
))
; ---------------------------
(DEFUN REMOVEF (LAMBDA (X LST)
   (COND ( (NULL LST) NIL )
         ( (EQ X (CAR LST)) (CDR LST) )
         (   T   (CONS (CAR LST) (REMOVEF X (CDR LST))) )
   )
))
```

Example 16. Counting the number of positive, negative and zero elements of the numeric list **LST**.

```
(DEFUN COUNT (LAMBDA (LST)
 ; Count the number of positive, negative, and
 ; zero elements of a numeric list LST
   (LIST (SUMMA LST PLUSP) (SUMMA LST ZEROP)
         (SUMMARY LST MINUSP))
))
; ---------------------------- %
( DEFUN SUM (LAMBDA ( LST FUNC )
; Count the number of elements in a numeric list
; for which the FUNC predicate is true
   (COND ( (NULL LST) 0 )
         ( (FUNC (CAR LST)) (+ 1 (SUMMA (CDR LST))) )
         (    T    (SUMMA (CDR LST)) )
   )
))
```

Example 17. Squaring the first odd element of a numeric list.

```
(DEFUN FIRST (LAMBDA (LST)
   (COND ( (NULL LST) NIL)
         ( (ODDP (CAR LST))
               (CONS (* (CAR LST) (CAR LST))
                     (CDR LST)) )
         (  T  (CONS (CAR LST) (FIRST (CDR LST))) )
   )
))
```

Example 18. Squaring all odd elements of a numeric list.

```
(DEFUN ALL (LAMBDA (LST)
   (COND ( (NULL LST) NIL)
         ( (ODDP (CAR LST))
               (CONS (* (CAR LST) (CAR LST))
                     (ALL (CDR LST))) )
         (  T  (CONS (CAR LST) (ALL (CDR LST))) )
   )
))
```

Example 19. Translating a natural number into "Roman notation". The function below implements the original Roman numeral system. Such shortened notations as IV instead of IIII came into use only at the beginning of the 16th century.

```
(DEFUN ROM (LAMBDA (N)
   (COND ( (NOT (< N 1000))
                    (CONS M (ROM (- N 1000))) )
         ( (NOT (< N  500))
                    (CONS D (ROM (- N  500))) )
         ( (NOT (< N  100))
                    (CONS C (ROM (- N  100))) )
         ( (NOT (< N   50))
                    (CONS L (ROM (- N   50))) )
         ( (NOT (< N   10))
                    (CONS X (ROM (- N   10))) )
         ( (NOT (< N    5))
                    (CONS V (ROM (- N    5))) )
         ( (NOT (< N    1))
                    (CONS I (ROM (- N    1))) )
   )
))
```

Example 20. Calculating the value of a polynomial of degree **N** using Horner 's scheme with coefficients stored in the list **COEF** at point **X.**

```
(DEFUN HORNER (LAMBDA (COEF N X)
   (COND ( (EQ N 0) (NTH 1 COEF) )
         (   T   (+ (NTH 1 COEF)
                     (* X (HORNER (CDR COEF)
                                  (- N 1)
                                   X)
                     )
                 )
         )
   )
))
; -----------------------
(DEFUN NTH (LAMBDA (N LST)
; The function returns the N-th element of the list LST
   (COND ( (EQ N 1) (CAR LST) )
         (    T     (NTH (- N 1) (CDR LST) ) )
   )
))
```

In the next step we will look at ***recursion by value***.

Step 37.
Simple recursion. Recursion by value

In this step we will look at examples of using recursion by value.

We will talk about ***recursion by value*** if a function call is an expression that defines the function's result.

Example 1. The predicate **MEMBER1(A L)** returns that part of the list **L** in which the sought atom **A** is the first element.

```
(DEFUN MEMBER1 (LAMBDA (A L)
   (COND ( (NULL L) NIL )
         ( (EQ (CAR L) A) L )
         ( T (MEMBER1 A (CDR L)))
   )
))
```

Example 2. A list is an asymmetric data structure that is simply traversed from left to right. In many cases, it is more natural to solve a problem by computing from right to left. For example, the same list reversal would be much easier to perform if the last element of the list could be accessed directly. This contradiction between the data structure and the problem-solving process leads to programming difficulties and can be a source of inefficiency.

In imperative programming languages, there is the possibility of using auxiliary variables in which intermediate results can be stored. In functional programming, variables are not used in this way. But the corresponding mechanism can be easily implemented using an auxiliary function in which the necessary auxiliary variables are parameters.

Then for the **REVERSE** function we get the following definition:

```
(DEFUN REVERSE (LAMBDA (L)
   (TRANCE L NIL)
))
; ----------------------------
(DEFUN TRANCE (LAMBDA (L Result)
   (COND ( (NULL L) Result )
         ( T (TRANCE (CDR L) (CONS (CAR L) Result)) )
   )
))
```

The auxiliary function **TRANCE** is recursive ***in value***. It transfers elements so that at each step of the recursion, the next element moves from the argument **L** to the argument **Result**.

Example 3. A very common criticism of the **LISP** language is that finding the k-th element of a list is a very long process. Let us give a function that takes as arguments a positive integer **I** and a list **LST**, and returns the element of the list with number **I**. It is assumed that the list **LST** has a length not less than **I**.

```
(DEFUN NTH (LAMBDA (I LST)
   (COND ( (EQ I 1) (CAR LST) )
         ( T (NTH (- I 1) (CDR LST)) )
   )
))
```

Example 4. Defining a predicate similar to the **MEMBER** predicate for multi-level lists.

```
   (DEFUN MEMBER2 (LAMBDA (X LST)
      (COND ( (NULL LST) NIL)
            (T(OR(COND((ATOM(CAR L)))
                                  (EQ X (CAR L)) )
                            ( T (MEMBER2 X (CAR L)))
                      )
                      (MEMBER2 X (CDR L))
                  )
            )
      )
   ))
```

Example 5. Constructing a list congruent to the list **LST** and containing the "depths of immersion" of elements in the list **LST**. For example, if **LST = ((1 (2) 3) 4)**, then the function returns a list of the form **((2 (3) 2) 1)**.

```
   (DEFUN COPY (LAMBDA (LST M)
      (COND ( (NULL LST) NIL )
            ( (ATOM LST) (POISON LST M) )
            ( T (CONS (COPY (CAR LST) M)
                      (COPY (CDR LST) M)) )
      )
   ))
   ; ------------------------
   (DEFUN POISK (LAMBDA (X LST)
   ; Finding the "depth of immersion" of X in the list LST
      (COND ( (MEMBER X LST) 1 )
            ( (MEMBER2 X (CAR LST))
                  (+ 1 (POISK X (CAR LST))) )
            ( T (POISK X (CDR LST)) )
      )
   ))
   ; --------------------------
   (DEFUN MEMBER2 (LAMBDA (X LST)
   ; The MEMBER2 predicate establishes the occurrence
   ; of element X in the multilevel list LST
      (COND ( (NULL LST) NIL)
            ( T (OR (COND ( (ATOM (CAR LST))
                                  (EQ X (CAR LST)) )
                            ( T (MEMBER2 X (CAR LST)) )
                      )
                      (MEMBER2 X (CDR LST))) )
      )
   ))
```

Example 6. Definition of a functional predicate **(ALL P L)** that is true if and only if the predicate **P,** which is a functional argument, is true for all elements of the list **L**.

Definition of a functional predicate **(EXISTS P L)** that is true when the predicate **P** is true for at least one element of the list **L**.

```
   (DEFUN ALL (LAMBDA (P L)
      (COND ( (NULL L) T )
            ( (P (CAR L)) (ALL P (CDR L)))
            ( T NIL )
      )
   ))
   ; ----------------------
   (DEFUN EXISTS (LAMBDA (P L)
      (COND ( (NULL L) NIL )
            ( (P (CAR L)) T )
            ( T (EXIST P (CDR L)) )
      )
   ))
```

Example 7. The famous American scientist Alonzo Church, the author of lambda notation and Church's thesis, created lambda calculus on the approaches to the theory of algorithms, which can now be considered nothing more than a theoretical model of modern functional programming. He struggled for many months to program the operation of subtracting one from a natural number in lambda calculus:

```
            0 - 1 --> 0
            (n+1) - 1 --> n
```

Church never managed to solve this problem and was already convinced of the incompleteness of his calculation, but in 1932 Stefan Kleene, then a young graduate student, proposed the following, at first glance artificial, but in fact fully corresponding to the essence of the matter construction [1, p.6]:

```
  F (x,y,z) = if x=0
                then 0
                otherwise if y+1=x
                         then z
                         otherwise F(x,y+1,z+1)
  Then n-1 = F(n,0,0).
```

Here is a function that solves this problem:

```
   (DEFUN DIFUNIT (LAMBDA (X Y Z)
      (COND ( (EQ X 0) 0 )
            ( (EQ (+ Y 1) X) Z )
            ( T (DIFUNIT X (+ Y 1) (+ Z 1)) )
      )
   ))
```

Example 8. Modeling the addition ***of integers*** using the identity: **(a + 1) + (b - 1) = a + b**.

```
   (DEFUN PLUS1 (LAMBDA (A B)
      (COND ( (EQ B 0) A )
            ( (PLUSP B) (PLUS1 (+ A 1) (- B 1)) )
            ( T (PLUS1 (- A 1) (+ B 1)) )
      )
   ))
   ; ----------------------
   (DEFUN PLUS2 (LAMBDA (A B)
   ; Modeling the addition of natural numbers
      (COND ( (EQ B 0) A )
            ( (PLUSP B) (PLUS2 (+ A 1) (- B 1)) )
            ( T NIL )
      )
   ))
```

Example 9. A function that returns the first atom of a list.

```
   (DEFUN FIRSTATOM (LAMBDA (L)
      (COND ( (ATOM L) L )
            ( T (FIRSTATOM (CAR L)) )
      )
   ))
```

Example 10. Removing the first **N** elements of a single-level list **LST**.

```
   (DEFUN DELETEN (LAMBDA (N LST)
      (COND ( (EQ N 0) LST )
            ( T (DELETEN (- N 1) (CDR LST)) )
      )
   ))
```

Example 11. The **LAST** function returns the last element of a list.

```
(DEFUN LAST (LAMBDA (L)
   (COND ( (NULL L) NIL )
         ( (NULL (CDR L)) (CAR L) )
         ( T (LAST (CDR L)) )
   )
))
```

Example 12. A predicate that tests whether its argument is a list (possibly empty) composed only of atoms.

```
(DEFUN ATOMLIST (LAMBDA (X)
   (COND ( (NULL X) T )
         ( (ATOM X) NIL )
         ( (ATOM (CAR X)) (ATOMLIST (CDR X)) )
         ( T NIL )
   )
))
```

Example 13. The **APPEND** function, which combines two lists into one list, and:

- ***recursion on arguments*** is used:

```
(DEFUN APPEND (LAMBDA (X Y)
   (COND ( (NULL X) Y )
         ( T (CONS (CAR X) (APPEND (CDR X) Y)) )
   )
))
```

- ***recursion by value*** is used:

```
(DEFUN APPEND (LAMBDA (X Y)
   (COND ( (NULL X) Y )
         ( T (APPEND (CDR X) (CONS (CAR X) Y)) )
   )
))
```

Note: As you can see, ***APPEND*** *copies the list that is the first argument. This function is often used in the form* ***(APPEND LST NIL)*** *when you want to make a copy of the top level of a list.*

Note that if the list ***X*** *is very long, the calculation will be slow. Creating list cells with the* ***COND*** *function takes time and adds work to the garbage collector in the future. If, for example, list* ***X*** *contains 1000 elements and list* ***Y*** *contains one element, then 1000 new cells will be created during the calculation, although it is only a question of adding one element to the list. If the order of the arguments were different, one cell would be created, and the lists would be combined approximately 1000 times faster.*

If it is not important for us that the value of variable ***X*** *will change, then we can use the "faster" function* ***NCONC instead of the APPEND*** *function. The* ***NCONC*** *function does the same thing as the* ***APPEND*** *function, with the only difference being that it simply concatenates the lists, changing the pointer in the* ***CDR*** *field of the last cell of the list, which is the first argument, to the beginning of the list, which is the second argument. By using the* ***NCONC*** *function, we avoid copying the list* ***X****, but as a side effect, the value of variable* ***X has changed. All other structures that used the cells of the original value of variable X*** *have also changed.*

(1) Henderson P. Functional programming: Application and implementation. - M.: Mir, 1983. - 349 p.

In the next step we will look at ***recursion by value and by arguments***.

Step 38.
Simple recursion. Recursion by value and by arguments

In this step we will look at examples of using recursion by value and by arguments.

In this section we will look at just a few examples.

Example 1. The **REMOVE** function returns a list in which all occurrences **of ATM** in the list **LST** have been removed.

```
(DEFUN REMOVE (LAMBDA (ATM LST)
   (COND ( (NULL LST) NIL)
         ( (EQ ATM (CAR LST))
               ; Recursion by value
               (REMOVE ATM (CDR LST)) )
         ( T   ; Recursion on argument
               (CONS (CAR LST)
                     (REMOVE ATM (CDR LST))) )
   )
))
```

The list **LST** is reduced by removing all identical **ATM** in the **EQ** sense elements (second branch) and copying the remaining elements (third branch) to the result list (**CONS**) until the termination condition (first branch) becomes true. The result is formed during the return process.

*Note: The **REMOVE** function can be defined using the **EQUAL** predicate instead of **EQ**. This function can remove elements that are lists!*

Example 2. The **MEMBERN** predicate allows one to determine the inclusion of a list **X** in a list **LST** at any level.

```
(DEFUN MEMBERN (LAMBDA (X LST)
   (COND ( (NULL LST) NIL)
         (  T  (OR (COND ( (ATOM (CAR LST))
                                   (MEMBERN X (CDR LST)) )
                         (  T  (OR (EQUAL X (CAR LST))
                                   (MEMBERN X (CAR LST))) )
                   )
                   (MEMBERN X (CDR LST))
               )
         )
   )
))
```

Example 3. The FOOT and **HEAD** functions given in the example below are similar to the standard Lisp functions **CAR** and **CDR**, but are designed to traverse a list ***from end to beginning***.

```
(DEFUN SUMMA (LAMBDA (X LST)
 ; Calculate the sum of the places where element X occurs in list LST ;
   (COND
      ( (NULL LST) 0  )
      ( (EQ X (FOOT LST))
               (+ (SUMMA X (HEAD LST)) (LENGTH LST)) )
      (    T   (SUMMA X (HEAD LST))                   )
   )
   ))
   ; ---------------------
   (DEFUN FOOT (LAMBDA (LST)
      (CAR (REVERSE LST))
   ))
   ; ---------------------
   (DEFUN HEAD (LAMBDA (LST)
      (REVERSE (CDR (REVERSE LST)))
   ))
```

Example 4.

```
   (DEFUN REVERSE1 (LAMBDA (LST)
    ; "Reversal" of the list LST at the first level
      ( (NULL LST) NIL)
      ( APPEND1 (REVERSE1 (CDR LST)) (CAR LST) )
   ))
   ; --------------------------
   (DEFUN APPEND1 (LAMBDA (LST X)
      ( (NULL LST)
             (CONS X NIL) )    ; Create a single-element list
      ( CONS (CAR LST) (APPEND1 (CDR LST) X) )
   ))
```

In the next step we will look at ***parallel recursion***.

Step 39.
Parallel recursion

In this step we will introduce the concept of parallel recursion and illustrate its use.

Recursion is called ***parallel*** if it occurs simultaneously in several arguments of a function.

"Here are two rules that can be used to decide whether to use recursion:

1. If a problem can be solved iteratively, then use iteration.
2. If the recursive version of a procedure contains two references to its own name, then it may be useful." [1, p.29].

Example 1. A function that calculates the number of atoms in a given list **LST** at all levels:

```
   (DEFUN  COUNT (LAMBDA (LST)
      (COND ( (NULL LST) 0 )
            ( (ATOM LST) 1 )
            (   T   (+ (COUNT (CAR LST))
                       (COUNT (CDR LST))) )
      )
   ))
```

Note that if you apply this function to a list of lists, the **NIL** elements that enclose each list will also be counted!

Example 2. Determining the number of non-**NIL** atoms in a given **LST** multi-level list.

```
   (DEFUN COUNTATOMS (LAMBDA (LST)
      (COND ( (NULL LST) 0 )
            ( (ATOM LST) 1 )
            (   T    (+ (COUNTATOMS (CAR LST))
                       (COUNTATOMS (CDR LST))) )
      )
   ))
```

Example 3. A function that computes the “depth” of a list. The phrase ***“depth” of a list*** is a special case of the concept of parenthesis level. To find the depth of a list for some atom, we count the number of left and right parentheses before that atom, and subtract the second from the first. Thus, in **((1 6) 7 ((8 4) 3))** the depth of the list for atom 8 is three.

```
   (DEFUN DEPTH (LAMBDA (X)
      (COND ( (ATOM X) 1 )
            (   T   (MAX (+ 1 (DEPTH (CAR X)))
                              (DEPTH (CDR X))) )
      )
   ))
   ; --------------------
   (DEFUN MAX (LAMBDA (X Y)
       (COND ( (> X Y) X )
             (   T    Y  )
       )
   ))
```

Example 4. The **COPY** function returns a copy of its argument.

```
   (DEFUN COPY (LAMBDA (EXPN)
      (COND ( (ATOM EXPN) EXPN )
            (   T  (CONS (COPY (CAR EXPN))
                         (COPY (CDR EXPN))) )
      )
   ))
```

Example 5. Find the sum of the elements of a multi-level numeric list.

```
   (DEFUN ADD (LAMBDA (L)
      (COND ( (NULL L) 0 )
            ( (ATOM (CAR L)) (+ (CAR L) (ADD (CDR L))))
            (   T   (+ (ADD (CAR L))
                       (ADD (CDR L))) )
      )
   ))
```

Example 6. The predicate **MEMBER3** sets the entry of element **X** into the multilevel list **LST** (if successful, returns the remainder of the list).

```
   (DEFUN MEMBER3 (LAMBDA (X LST)
      (COND ( (NULL LST) NIL)
            ( (COND ( (ATOM (CAR LST))
                          (COND ( (EQ X (CAR LST)) LST )
                                (  T  (MEMBER3 X (CDR LST)) )) )
                    (  T  (MEMBER3 X (CAR LST)) )
              )
            )
            (  T  (MEMBER3 X (CDR LST)) )
      )
   ))
```

Example 7. Let's consider a function that is designed to reverse the order of elements of a list **L** and its sublists, regardless of their location and nesting depth.

The SUPERREVERSE function reverses the head of a list, forms it into a list, and appends a reversed tail to the list at the front.

```
   (DEFUN SUPERREVERSE (LAMBDA (L)
      (ATOM L) L)
      ( (NULL (CDR L)) (CONS (SUPERREVERSE (CAR L)) NIL) )
      ( APPEND (SUPERREVERSE (CDR L))
               (SUPERREVERSE (CONS (CAR L) NIL)) )
   ))
   ; -----------------------
   (DEFUN APPEND (LAMBDA (X Y)
   ; The APPEND function returns a list consisting of
   ; the elements of list X appended to list Y
      (COND ( (NULL X) Y )
            (   T   (CONS (CAR X) (APPEND (CDR X) Y)) )
      )
   ))
```

Example 8. “Compress” the list structure into a single-level list, i.e. remove all nested brackets.

```
   (DEFUN ONE-RANGE (LAMBDA (LST)
      ( (NULL LST) NIL )
      ( (ATOM LST) (CONS (CAR LST) NIL) )
      ( APPEND (ONE-RANGE (CAR LST)) (ONE-RANGE (CDR LST)) )
   ))
   ; -----------------------
   (DEFUN APPEND (LAMBDA (X Y)
   ; The APPEND function returns a list consisting of
   ; the elements of list X appended to list Y
      (COND ( (NULL X) Y )
            (   T   (CONS (CAR X) (APPEND (CDR X) Y)) )
      )
   ))
```

The ONE-RANGE function combines (using the **APPEND** function) the head of a list "compressed" into one level and the "compressed" tail. If the head of the list is an atom, then a list is formed from it, since the arguments of the **APPEND** function must be lists.

Example 9. Modeling the **EQUAL** predicate using the **EQ** predicate.

```
   (DEFUN EQUAL1 (LAMBDA (L M)
      (COND ( (NULL L) (NULL M) )
            ( (ATOM L) (AND (ATOM M) (EQ L M)) )
            ( (ATOM M) NIL )
            (  T  (AND (EQUAL1 (CAR L) (CAR M))
                       (EQUAL1 (CDR L) (CDR M))) )
      )
   ))
```

Example 10. Construct a predicate **LISTP** that returns the value **NIL** if the given expression is an atom other than **NIL** or an expression that can be written only in dot notation. Otherwise, the given expression is a list that can be written without resorting to dot notation at any level, and the predicate returns the value **T**.

```
   (DEFUN LISTP (LAMBDA (X)
      (COND ( (NULL X) T )
            ( (ATOM X) NIL )
            ( (OR (ATOM (CAR X)) (LISTP (CAR X)))
                                 (LISTP (CDR X)) )
            ( T  NIL )
      )
   ))
```

Test examples:

```
   $ (LISTP '((A B) (B. C) C))
   NIL
   $ (LISTP '((A . (B C)) (B . (C A))))
   T
```

Example 11. Recursive methods are best justified in problems in which formal and, above all, natural recursion is encountered in structures and processes.

We implement the classic oriental game “Towers of Hanoi”. The game consists of the following. Three vertical rods **A, B** and **C** and a set of **N** round disks of different sizes with a hole are used. In the initial state, the disks are strung in order according to their sizes on rod **A.**

The goal is to transfer all the disks to the rod **B.** However, the disks are not transferred in any random order; the following rules must be followed when transferring:

1. You can only transfer one disk at a time.
2. A larger disk cannot be placed on a smaller one.

The third rod **C** can be used as an auxiliary (intermediate) one. If it is free or there is a larger disk there, then the next disk can be moved onto it while the disk lying below is being moved. This idea is the solution to the game.

Algorithm for the Towers of Hanoi problem:

1. ***Transfer N-1 disks from rod A to auxiliary rod C (Towers of Hanoi problem for N-1).***
2. ***Move the bottom disk from rod A to rod B.***
3. ***Transfer N-1 disks from rod C to rod B (Towers of Hanoi problem for N-1).***

```
(DEFUN HANOI (LAMBDA (NDISKS)
   (MOVE-DISK A B C NDISKS) (QUOTE Done!)
))
; -------------------------------------
(DEFUN MOVE-DISK (LAMBDA (SRC DEST AUX N)
   ( (EQ N 1)(PRIN1 SRC)(PRIN1 " --> ") (PRINT DEST) )
   ( (MOVE-DISK SRC AUX DEST (- N 1))
             (PRIN1 SRC)(PRIN1 " --> ") (PRINT DEST)
     (MOVE-DISK AUX DEST SRC (- N 1))
   )
))
```

Test example:

```
   $ (HANOI 3)
   A --> B
   A --> C
   B --> C
   A --> B
   C --> A
   C --> B
   A --> B
   Done!
```

The problems for towers consisting of one or two disks are solved easily. The problem for three disks requires more transfers.

Is it possible to find ***a non-recursive solution*** to the Tower of Hanoi problem? The answer is yes. Although its justification is much more complicated than the recursive solution.

A direct solution was found by P. Bueneman and L. Levy. The algorithm sequentially performs the following actions [2, p.64-65].

1. Move the smallest disk from the rod it is currently on to the rod next in clockwise order.
2. Move any disk except the smallest one.

---

(1) Foster J. List Processing. - M.: Mir, 1974. - 72 p.
(2) Anisimov A.V. Computer Science. Creativity. Recursion. - Kiev: Nauk.dumka, 1988. - 224 p.

In the next step we will look at ***mutual recursion***.

Step 40.
Mutual recursion

In this step we will introduce the concept of mutual recursion and illustrate its use.

If two or more functions call each other, the recursion is called ***mutual***. Unfortunately, we were able to find only one meaningful example of mutual recursion!

Example. A function that determines whether the number of minuses in a given list **LST** containing only + and - atoms is odd. In [1, p.149] it is stated that in this example one of the functions cannot be eliminated by substitution.

```
(DEFUN ISPOS (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (AND (NULL (CDR LST)) (EQ (CAR LST) -)) T )
         ( (EQ (CAR LST) -) (ISNEG (CDR LST)) )
         ( (EQ (CAR LST) +) (ISPOS (CDR LST)) )
   )
))
;------------------------
(DEFUN ISNEG (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (AND (NULL (CDR LST)) (EQ (CAR LST) +)) T )
         ( (EQ (CAR LST) -) (ISPOS (CDR LST)) )
         ( (EQ (CAR LST) +) (ISNEG (CDR LST)) )
   )
))
```

Note. ***On programming nested loops***. *The multiple repetitions corresponding to the nested loops of imperative programming in functional programming are usually implemented using two or more functions, each of which corresponds to a simple loop. The call of such a recursive function is used in the definition of another function as an argument of its recursive call.*

---

(1) Bauer F.L., Goos G. Computer Science. Introductory Course: In 2 parts. Part 1. translated from German. - Moscow: Mir, 1990. - 336 p.; Part 2. translated from German. - Moscow: Mir, 1990. - 423 p.

In the next step we will talk about ***higher order recursion***.

Step 41.
Higher-order recursion

In this step we will introduce the concept of higher-order recursion and illustrate its use.

The expressive possibilities of recursion are already evident from the meaningful and space-saving definitions given in the previous steps.

By using increasingly powerful types of recursion, more complex calculations can be written in relatively concise ways. At the same time, the complexity of programming and understanding the program increases.

Let us consider nested loop programming in such a form that in the definition of a function the recursive call is an argument to the call of the same function. In such a recursion, we can distinguish different orders according to the level of recursion at which the call is located.

We will call this form of recursion ***higher-order*** recursion. The functions we have defined so far have been functions with ***zero-order*** recursion.

Example 1. A well-known example of higher-order recursion from the theory of recursive functions ***is the Ackermann function***, which is often cited as a "bad" function: it is a function with ***first-order recursion***, and the computation time grows exponentially even for small values of the argument.

The two-place Ackermann-Hermes function f of non-negative integer arguments is defined by the relations:

```
f(0,b) = b+1,
f(a,0) = f(a-1,1) for a>0,
f(a,b) = f(a-1,f(a,b-1)) for a>0 or b>0.
```

Let's write a function that calculates its value:

```
(DEFUN AKKERMAN (LAMBDA (M N)
   (COND ( (EQ M 0) (+ N 1) )
         ( (EQ N 0) (EQUALITY (- M 1) 1) )
         ( T (AKKERMAN (- M 1)
                          (AKKERMAN M (- N 1))) )
   )
))
```

Let's select ***test examples***.

```
                          ******************
1) Let us prove that * f(1,b) = b+2 *
                          ******************
f(1,b) = f(0,f(1,b-1)) = 1+f(1,b-1) = 1+f(0,f(1,b-2)) =
       = 1+1+f(1,b-2) = 2+f(1,b-2) =...=
         { We apply this process b times }
       = b+f(1,0) = b+f(0,1) = b + 2, ч.т.д.
                          *******************
2) Let us prove that * f(2,b) = 2b+3 *
                          *******************
         f(2,b) = f(1,f(2,b-1)) =
         { As previously proven } =
         2 + f(2,b-1) =
       = 2+f(1,f(2,b-2)) = 2+2+f(2,b-2) = 2+2+f(1,f(2,b-3)) =
       = 2 + 2 + 2 + f(2,b-3) =
             { Apply this process b times } =
           = 2b+f(2,0) = 2b+f(1,1) = 2b + 3, ч.т.д.

   3) Let's prove that *******************
                         * f(3,b) = 2^{b+3}-3 *
                         *******************
   f(3,b) = f(2,f(3,b-1)) = 2f(3,b-1)+3 = 2[f(2,f(3,b-2))]+3 =
          = 2[2f(3,b-2)+3]+3 = 2[2f(2,f(3,b-3))+3]+3 =
          = 2{2[2xf(3,b-3)+3]+3}+3 =
          = 2*2*2*f(3,b-3) + 2*2*3 + 2*3 + 3 =
          = { After repeating this operation b times } =
          = 2^b*f(3,0) + 2^{b-1}*3 + 2^{b-2}*3 + 2^{b-1}*3 +...+ 3 =
                           1-2^b
          = 2^b*f(3,0) + 3*---  = 2^b*f(3,0) + 3*2^b -3.
                           1-2
   Now let's calculate
          f(3,0) = f(2,1) = f(1,f(2,0)) = 2 + f(2,0) =
                   2 + f(1,1) = 2+3 = 5

   f(3,b) = 5*2^b+3*2^b-3 = 8*2^b-3, ч.т.д.

   4) Let's move on...

          f(4,b) = f(3,f(4,b-1)) = 2^{f(4,b-1)+3}-3
   If we set f(4,b)=H(b)-3, then the function H will satisfy
functional equation of "hyperpower":
                           H(b) = 2 ^{H(b-1)}
c H(0) = 3+f(4,0) = 3 + f(3,1) = 3+(2 ^4 -3) = 16
   Thus,
  H(1) = 2^{H(0)} = 2^{16} = 3+f(4,1) --> f(4,1) = 2^{16}-3
```

Remark 1. The idea of a function including a sum, product, power, hyperpower, etc. goes back to ***Wilhelm Ackermann*** *(1896-1962). In 1926, D. Hilbert posed the problem of whether any computable function could be obtained in some simple way - essentially by a finite application of the processes of substitution and induction ("primitively recursively"). Ackermann showed by his example in 1928 that this was not so. For each fixed* ***a,*** *the Ackermann-Hermes function will, of course, be primitively recursive.*

The Ackermann function was proposed to prove that non-primitive recursive functions exist.

Note 2. Compare the given program with a program written in the classical imperative programming language ***Pascal****:*

```
   PROGRAM  A_k_k_e_r_m_a_n;
   var  M,N: Integer;
  { -------------------------------------- }
   FUNCTION  A_k_k (M,N: Integer): Integer;
   BEGIN
      If  M=0 then  A_k_k := N + 1
      else  If  N=0 then  A_k_k := A_k_k (M-1,1)
            else  A_k_k := A_k_k (M-1,A_k_k (M,N-1))
   END;
  { --- }
   BEGIN
      ReadLn (M,N); Writeln; Writeln (A_k_k (M,N))
   END.
```

Example 2. Write a function that implements the calculation of the value of the Ackermann function using the following formulas:

```
                     X+1                ,  N=0
```

```
                    X, N=1, Y=0
      A(N,X,Y) = 0, N=2, Y=0
                    1, N=3, Y=0
                    2, N>=4, Y=0
                    A(N-1,A(N,X,Y-1),X), N<>0, Y<>0
```

It is interesting to note that:

```
          A(0,X,Y) = X+1; A(1,X,Y) = X+Y;
          A(2,X,Y) = X*Y; A(3,X,Y) = X Y.
(DEFUN AKKER (LAMBDA (N X Y)
   (COND ( (EQ N 0) (+ X 1) )
         ( (AND (EQ N 1) (EQ Y 0)) X )
         ( (AND (EQ N 2) (EQ Y 0)) 0 )
         ( (AND (EQ N 3) (EQ Y 0)) 1 )
         ( (AND (EQ N 4) (EQ Y 0)) 2 )
         ( (AND (> N 4) (EQ Y 0)) 2 )
         ( T (AKKER (- N 1)
                    (AKKER NX (- Y 1))
                     X) )
   )
))
```

Example 3. Calculate for a given N the value of the function given by the formula: **F(N) = If N>202 then N-3 else F(F(N+4))**

```
(DEFUN RECURS1 (LAMBDA (N)
   (COND (> N 202) (- N 3)
         (   T   (RECURS1 (RECURS1 (+ N 4))) )
   )
))
```

Example 4. As another example of a function with first-order recursion, we give the **ONE-LEVEL** function, which places the elements of a list at the same level, which we defined earlier using parallel recursion:

```
(DEFUN ONE-LEVEL (LAMBDA (L)
   (INLINE L NIL)
))
; ----------------------------
(DEFUN INLINE (LAMBDA (L RESULT)
   ( (NULL L) RESULT )
   ( (ATOM L) (CONS L RESULT) )
   ( INLINE (CAR L) (INLINE (CDR L) RESULT) )
))
```

Note that by using higher-order recursion, the computation can be represented more abstractly and with a shorter definition, but it is quite difficult to visualize how such a function works.

The INLINE function works as follows. The result is built into the **RESULT** list. If **L** is an atom, it can be directly added to the beginning of the **RESULT** list. If **L** is a list and its first element is an atom, then everything is reduced to the previous state at the next level of recursion, but in a situation where the **RESULT** list already contains the remaining tail extended one level.

In the case where the head of the list **L** is also a list, it is first reduced to one level. This is done by recursive calls that dive into the head branch until an atom is encountered that can be added to the beginning of the structure that has been extended to one level by that point. The atoms encountered in this way are added one by one to the extended tail. At each level, when the list is exhausted, **the RESULT** that has been typed to that point is returned to the previous level.

Example 5. The following definition of the **REVERSE** function is an example of an even deeper level of recursion:

```
(DEFUN REV (LAMBDA (L)
   ( (NULL L) L )
   ( (NULL (CDR L)) L )
   ( CONS (CAR (REV (CDR L)))
          (REV (CONS (CAR L)
                     (REV (CDR (REV (CDR L)))))) )
))
```

The definition uses second-order recursion. The computations represented by this definition are more difficult to understand than the previous ones, and the complex recursion makes the computations longer.

Example 6. The **SUPERREVERSE** function returns a list of elements from **LST1** reversed at all levels.

```
(DEFUN SUPERREVERSE (LAMBDA (LST1 LST2)
   ( (NULL LST1) LST2 )
   ( (ATOM (CAR LST1) )
       (SUPERREVERSE (CDR LST1) (CONS (CAR LST1) LST2)))
   ( SUPERREVERSE (CDR LST1)
                  (CONS (SUPERREVERSE (CAR LST1)) LST2) )
))
```

Example 7. "Fast" version of the **MOD** function for calculating the remainder in integer division.

```
(DEFUN MOD1 (LAMBDA (M N)
   (COND ( (NOT (LESSP M (TIMES 2 N)))
                    (MOD1 (MOD1 M (TIMES 2 N)) N) )
         ( (NOT (LESSP M N))
                    (MOD1 (- M N) N) )
         (  T   M  )
   )
))
; --------------------
(DEFUN MOD (LAMBDA (M N)
; MOD function for calculating the remainder
; in integer division
   (COND ( (LESSP M N) M )
         (   T   (MOD (- M N) N) )
   )
))
```

Higher-order recursion forms are not usually used in practical programming, but they have their own theoretical and methodological significance.

In the next step we will introduce the concepts ***of auto-application and auto-replication***.

Step 42.
Auto-application and auto-replication

In this step we will introduce the concepts of auto-application and auto-replication.

A functional that receives itself as an argument is called ***an autoapplicative function (applied to itself)***. Accordingly, a function that returns itself is called ***an autoreplicative (self-reproducing) function***.

Autoapplicative and autoreplicative functions form ***a class of autofunctions***. Of course, it is not obvious that such functions exist at all. However, there are no fundamental obstacles to their existence and definition. In the **LISP** language, they can be defined and used quite simply.

Example: Let's take as an example the simplest possible function that returns itself as a result:

```
   $ ((LAMBDA (X)         (LIST X (LIST (QUOTE QUOTE) X)))
      (QUOTE (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))) )
```

Result:

```
         ((LAMBDA (X)(LIST X (LIST (QUOTE QUOTE) X)))
   (QUOTE (LAMBDA (X)(LIST X (LIST (QUOTE QUOTE) X)))) )
```

The function is also autoapplicative! To check, let's assign it to the **SAMO** atom:

```
   $ (SETQ SAMO (QUOTE ((LAMBDA (X)   (LIST X (LIST (QUOTE QUOTE) X)))
                (QUOTE (LAMBDA   (X)   (LIST X (LIST (QUOTE QUOTE) X)))))))
```

Result:

```
   $ (EVAL (EVAL (EVAL SAMO)))
   (      (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))
   (QUOTE (LAMBDA (X) (LIST X (LIST (QUOTE QUOTE) X)))))
```

The value of a function can be calculated over and over again, and still get the same result: the function definition itself (a copy of it). In addition, the value of a function is the same as the value of the argument of its call. Such a value is called ***a fixed point of the function***.

Auto-application as a form of recursion is very interesting from a theoretical point of view, but in practical programming, at least up to now, it does not find much application. It is associated with theoretical questions and problems that are largely not yet studied.

However, auto-application can open up a new approach to programming. Possible applications could be, for example, systems that keep certain properties constant, although some parts of them change.

Autofunctions are a new class of functions that differ from ordinary recursive functions in the same way that recursive functions differ from non-recursive ones.

In the next step we will offer you to complete several tasks using recursion.

Step 43.
Practical lesson #2. Functional programming Recursion

In this step we will provide solutions to some problems using recursion.

The structure of your program text for execution using the **muLISP-85** interpreter:

```
   ; --------------------------------------------------------
   ; The condition of your task
   ; --------------------------------------------------------
   (DEFUN Your_function_name (LAMBDA (Your_function_arguments)
   function body
   ))
   (RDS)
```

To load your functions into the **muLISP-85** interpreter, use the **RDS** function as follows: **(RDS FileName.FileExtension)**.

Example 1.

```
(DEFUN SUMMA (LAMBDA (X Y)
   (+ X Y)
))
; ---------------------
(DEFUN ADD1 (LAMBDA (NUM)
; Increment function
   (+ NUM 1)
))
; ---------------------
(DEFUN SUB1 (LAMBDA (NUM)
; Decrement function
   (- NUM 1)
))
; ------------------------
(DEFUN PERCENT (LAMBDA (A B)
; The function returns A percent of the number B
   (/ (* A 100) B)
))
; --------------------
(DEFUN ABS (LAMBDA (NUM)
; The ABS function returns the absolute value of the NUM argument
   (COND ( (MINUSP NUM) (- NUM))
         ( T NUMBER )
   )
))
; --------------------
(DEFUN MAX (LAMBDA (M N)
; The MAX function returns the larger of two numbers
   (COND ( (> M N) M )
         ( T N )
   )
))
; ----------------------
(DEFUN PLUS (LAMBDA (X Y)
; Fragment of complex arithmetic
; X and Y are lists of the form (x y)
   (LIST (+ (CAR X) (CAR Y))
         (+ (CADR X) (CADR Y)) )
 ))
```

Example 2. Calculate the factorial of an integer **X.** The problem is solved using a function that takes a value equal to one when the argument is equal to one or zero, and in all other cases is equal to the product of the argument and the value of the same function of the argument, reduced by one. Solution:

```
(DEFUN FACT (LAMBDA (X)
   (COND ( (ZEROP X) 1 )
         (     T    (* X (FACT (- X 1))) )
   )
))
```

Let's depict the process ***of downward recursion*** graphically:

```
        (FACT 3)
        --------
            V
         (* 3 (FACT 2))
```

```
                 ---------
                     V
                (* 2 (FACT 1))
                     ---------
                         V
                   (* 1 (FACT 0))
                         ---------
                             V
                             1
```

Now let's depict the process of "exiting recursion":

```
     (* 3 (* 2 (* 1 1))) -> 6
```

Example 3. Calculate the value of the function given by the formula: **F(N) = If N>100 then N-10 else F(F(F(N+21)))**. Solution:

```
  (DEFUN FN (LAMBDA (N)
     (COND
        ( (> N 100) (- N 10) )
        (  T  (FN (FN (FN (+ N 21))))    )
     )
  ))
```

Test examples:

```
  $ (FN 94)
  91
  $ (FN 100)
  91
  $ (FN 123)
  113
```

Example 4. Define a **COPY** function that returns a copy of the given list. Solution:

```
  (DEFUN COPY (LAMBDA (LST)
   ; Function to copy the "top" level of the list LST
     (COND
        ( (NULL LST) NIL )
        (     T      (CONS (CAR LST) (COPY (CDR LST))) )
     )
  ))
```

Example 5. Define a function that removes the last element from a list. Solution:

```
  (DEFUN DROPLAST (LAMBDA (L))
     (COND
        ( (NULL L)       NIL                           )
        ( (NULL (CDR L)) NIL                           )
        (   T            (CONS (CAR L) (DROPLAST (CDR L))) )
     )
  ))
```

Example 6. Construct a predicate that checks whether its argument is a list (possibly empty) composed only of atoms. Solution:

```
    (DEFUN ATOMLIST (LAMBDA (X)
      (COND
         ( (NULL X)       T                    )
         ( (ATOM X)       NIL                  )
         ( (ATOM (CAR X)) (ATOMLIST (CDR X)) )
         (  T NIL                              )
      )
    ))
```

Example 7. Define a function that removes the first occurrence of a given element at the top level from a list. Solution:

```
   (DEFUN DELETEFIRST (LAMBDA (A L)
      (COND
         ( (NULL L) NIL )
         ( (EQUAL (CAR L) A) (CDR L)                    )
         (  T (CONS (CAR L) (DELETEFIRST A (CDR L))) )
      )
   ))
```

Example 8. Define a function that removes every second element from a list. Solution:

```
   (DEFUN EVERYSECOND (LAMBDA (L)
      (COND
         ( (NULL L) NIL )
         ( (NULL (CDR L)) L                          )
         (  T (CONS (CAR L) (EVERYSECOND (CDDR L))) )
      )
   ))
```

Example 9. Determine the largest element of a single-level numeric list **LST**. Solution:

```
   (DEFUN MAX (LAMBDA (LST)
      (COND
         ( (NULL LST) NIL                        )
         ( (EQ (LENGTH LST) 1)   (CAR LST)       )
         (  T (MAX2 (CAR LST) (MAXIM (CDR LST))) )
      )
   ))
   ; ---------------------
   (DEFUN MAX2 (LAMBDA (X Y)
      (COND
         ( (> X Y) X )
         (  T Y      )
      )
   ))
```

What corrections need to be made in the program to obtain the smallest element?

Example 10. Define a function that returns a list whose first element is the sum and the second element is the product of the elements of the given list **LST**. Solution:

```
(DEFUN SP (LAMBDA (LST)
   (LIST (SUM LST) (PR LST))
))
; ---------------------
(DEFUN SUM (LAMBDA (LST)
; Sum of elements of a numeric list
   (COND
      ( (NULL LST) 0                       )
      (  T (+ (CAR LST) (SUM (CDR LST))) )
   )
))
; --------------------
(DEFUN PR (LAMBDA (LST)
; Product of elements of numeric list LST
   (COND
      ( (NULL LST) 1                      )
      (  T (* (CAR LST) (PR (CDR LST))) )
   )
))
```

Test examples:

```
$ (SP '(1 2 3 4))
(10 24)
$ (SP '())
(0 1)
$ (SP '(1))
(1 1)
$ (SP '(0 1))
(1 0)
```

Example 11. Determine the number of elements equal to the maximum element of a numeric list. Solution:

```
(DEFUN AAA (LAMBDA (LST)
   (KOL LST (MAXIM LST))
))
; ----------------------
(DEFUN COL (LAMBDA (LST M)
   (COND
      ( (NULL LST) 0                            )
      ( (EQ (CAR LST) M) (ADD1 (KOL (CDR LST) M)) )
      (  T (KOL (CDR LST) M)                    )
   )
))
; ----------------------
(DEFUN MAX (LAMBDA (LST)
; The function returns the maximum element of the list LST
   (COND
      ( (NULL LST) NIL                      )
      ( (EQ (LENGTH LST) 1)  (CAR LST)       )
      (  T (MAX2 (CAR LST) (MAXIM (CDR LST))) )
   )
))
; ---------------------
(DEFUN MAX2 (LAMBDA (X Y)
; The function returns the larger of two numbers X and Y %
   ( ((> X Y) X ) Y )
))
; ---------------------
```

```
   (DEFUN ADD1 (LAMBDA (NUM)
   ; Function that increases the argument by 1
      (+ NUM 1)
   ))
```

Test examples:

```
   $ (AAA '(1 2 3 4 5))
   1
   $ (AAA '(2 2 3 4 5))
   2
   $ (AAA '(5 5 5 5 5))
   5
   $ (AAA '())
   0
```

Example 12. In a numerical list, find the largest element and the number of the first such element, if there are several. Solution:

```
   (DEFUN MAIN (LAMBDA (LST)
      (POSITION (MAXIM LST) LST)
   ))
   ; ----------------------
   (DEFUN MAX (LAMBDA (LST)
   ; The function returns the largest element of the list
      (COND
         ( (NULL LST) NIL                         )
         ( (EQ (LENGTH LST) 1) (CAR LST)          )
         (  T (MAX2 (CAR LST) (MAXIM (CDR LST))) )
      )
   ))
   ; ---------------------
   (DEFUN MAX2 (LAMBDA (X Y)
   ; The function returns the largest of two numbers X and Y
      (COND
         ( (> X Y) X )
         (  T Y      )
      )
   ))
   ; ----------------------------
   (DEFUN  POSITION (LAMBDA (X LST)
   ; The POSITION function returns the position of the atom X in
   a single-level list LST (the first element has
   ; number 1). If the element is not in the list, the function
   ; returns 0.
      (COND
         ( (NULL LST)       0 )
         ( (EQ X (CAR LST)) 1 )
         ( (MEMBER X LST)
              (+ 1 (POSITION X (CDR LST))) )
         ( T  0 )
      )
   ))
```

Test examples:

```
   $ (MAIN '(6 8 3 5 8 4 2 8))
   2
   $ (MAIN '(6 1 3 5 8 4 2 8))
   5
```

Example 13. Construct a function that removes from a list all elements that match a given atom (in the sense of **EQ**) and returns a list of all remaining elements. Solution:

```
(DEFUN REMBER (LAMBDA (X LST)
   (COND
      ( (NULL LST) NIL                             )
      ( (EQ X (CAR LST)) (REMBER X (CDR LST))      )
      (  T (CONS (CAR LST) (REMBER X (CDR LST))) )
   )
))
```

Test examples:

```
$ (REMBER 5 '(4 6 5 7 2 5 9 5 87))
(4 6 7 2 9 87)
$ (REMBER 12 '(1 86 12 6 7 12 0)
(1 86 6 7 0)
```

Example 14. Define a function **CDRN** such that **(CDRN N L)** is equivalent to a function **(CD...DR L)** whose name contains **N** letters **D**. Solution:

```
(DEFUN CDRN (LAMBDA (N LST)
   (COND
      ( (EQ N 0) LST                  )
      ( (EQ N 1) (CDR LST)            )
      (  T (CDNR (- N 1) (CDR LST)) )
   )
))
```

Test examples:

```
$ (CDNR 3 '(1 2 3 4 5))
(4 5)
$ (CDNR 3 '(1 2 (3 4 5)))
NIL
$ (CDNR 3 '(1 2 3 (4 5) 6))
((4 5) 6)
```

Example 15. Write a function that depends on three arguments **U, N** and **V**, inserting a list **V into a list U**, starting with the **N-th** element. For example:

```
$ (MAIN '(1 2 3) 2 '(4 5 6))
(1 2 4 5 6 3)
```

Solution:

```
(DEFUN MAIN (LAMBDA (LST1 N LST2)
   (COND
      ( (EQ (LENGTH LST1) N) (APPEND LST1 LST2) )
      (  T (APPEND (FIRSTN LST1 N) (APPEND LST2
                (LASTN LST1 (- (LENGTH LST1) N)))
           )
      )
   )
))
; -------------------------
(DEFUN FIRSTN (LAMBDA (LST N)
; Selects the first N elements of the list LST into a list
   (COND ( (EQ N 1) (LIST (CAR LST)) )
         ( T (CONS (CAR LST) (FIRSTN (CDR LST)
                                     (- N 1))))
   )
))
; ------------------------
(DEFUN LASTN (LAMBDA (LST N)
; Selects the last N elements of the list LST into the returned list
   (REVERSE (FIRSTN (REVERSE LST) N))
))
; -----------------------------
(DEFUN APPEND (LAMBDA (LST1 LST2)
; Makes a list from the first N elements of the list LST1
; and the inserted list LST2
   (COND ( (NULL LST1) LST2 )
         (   T  (CONS (CAR LST1)
                      (APPEND (CDR LST1) LST2) ))
   )
))
```

Tasks for independent solution

- Select from the numeric list **LST** all elements divisible by 3 and form a list from them.
- Calculate the number of positive, negative and zero elements in the numeric list **LST**.
- Describe a function that calculates the product of the elements of a numeric list.
- Describe a function that converts a numeric list into a list in which each element is one less.
- Define a function that returns a list whose first element is the sum of the squares and the second element is the product of the squares of the elements of the numeric list **LST**.
- Check if the given numeric list contains an element that is equal to the sum of the first and last elements.
- Define a function **(FIB N)** that computes **the N-th** element of the Fibonacci sequence. The Fibonacci numbers are defined as follows: $F_1 = 1$, $F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$ for n>=3.
- Define a function **COUNTG** such that **(COUNTG L N)** returns the number of number atoms in a list **L** greater than **N**. It is assumed that **L** has no sublists and that all its atoms are numbers. Thus, if **X** is (2 8 8 7 -4), then **(COUNTG X 7)** returns 2.
- In the list, swap the first and last items.
- Write a function that performs a cyclic permutation of the elements of a single-level list, in which the first element of the list becomes the last.
- Write a function to find those elements of a numeric list that are between integers **H** and **L.**
- Write a function **INRANGE** such that **(INRANGE V MAXIMUM MINIMUM)** returns **T** if **V** lies between **MAXIMUM** and **MINIMUM**, including the bounds, and **NIL** otherwise. It is assumed that **V** is a numeric atom, and **MAXIMUM** and **MINIMUM** are the largest and smallest elements of the numeric list, respectively.
- Find a function **(CDRN N L)** that is equivalent to a function **(CD...DR L)** whose name contains **N** letters **D**.
- Define a function **DOMINATE with two lists of EQUAL** values as arguments. The function **(DOMINATE L M)** should return **T** if each atom of **L** is greater than the corresponding atom of **M**, and **NIL** otherwise (all atoms are numeric).
- Write a function REVERSE2 that concatenates two lists, where both lists must be reversed. For example:

```
   $ (REVERSE2 ((A B) (C D)))
   (B A D C)
```

- Construct a function that removes from a numeric list all elements that match a given numeric atom and returns as its value a list of the squares of the remaining elements.
- Write a **DELETE** function that deletes **the N-th** element of the list **LST**.

- Write a function **INSLIST**, given three arguments **N**, **U**, and **V**, that inserts a reversed list **V** into a list **U** starting at the **N-th** element. For example:

```
    $ (INSLIST (2 (A B) (C D)))
    (A D C B)
```

- Calculate the product of the sums of positive and negative elements of a single-level numeric list **LST**.
- Check if there are any negative elements in the given numerical list. If there are, then construct a new list consisting of the negative elements of the original list, written in the same order.
- In a numeric list, each element is 0, 1, or 2. Rearrange the elements of the list so that all the zeros come first, then all the ones, and finally all the twos (you cannot sort the list!).
- Calculate the sum of the elements of a numeric list.
- Find the maximum element of a numeric list.
- Find the maximum element among the negative elements of a numeric list.
- Find the minimum element among the positive elements of a numerical list.
- Determine the number of elements equal to the minimum element of a numeric list.
- In a numeric list, you need to find the smallest element and the number of the first such element, if there are several.
- Perform a cyclic permutation of the list elements: the first element should become the second, the second - the third, etc., the last - the first.
- **Determine whether the sum of the elements of a numeric list is an even number.**
- For integers A,B,C>0, the function REM (A,B,C), which calculates the remainder of dividing A B by C, is defined recursively:

```
REM(A*A,B/2,C)  for A<C,B>2, B-even:
                MOD (A*REM(A*A,(B-1)/2,C)),
REM(A,B,C)      for A<C,B>=3, B is odd:
                A,                   for A<C, B=1;
                REM (MOD (A,C),B,C), for A>=C.
```

- Write a program to calculate **REM(A,B,C)**.
- Create a function that writes the elements of a list in reverse order to the given one.
- Define addition and multiplication operations on quaternions.
- Write a function to calculate the number of combinations of **n** elements by **m**.
- Write a function that calculates N!!.
- Define a function that returns the last atom of a list.
- Define a function that removes every third element from a list.

- Determine whether the elements in a single-level list containing Latin letters are sorted alphabetically.

In the next step we will talk about ***programming paradigms***.

Step 44.
Programming paradigms

T. Kuhn [1, p.11] writes: “By ***paradigms*** I mean universally recognized scientific achievements that, over a certain period of time, provide the scientific community with a model for posing problems and solving them.”

"The study of paradigms... is what chiefly prepares the student for membership in a scientific community. Since he thus associates himself with men who have learned the foundations of their field of study from the same concrete models, his subsequent practice in scientific research will not often show a sharp divergence from fundamental principles. Scientists whose scientific activity is based on the same paradigms rely on the same rules and standards of scientific practice. This community of attitudes and the apparent coherence they provide are the prerequisites for ***normal science***, that is, for the genesis and continuity in the tradition of a particular line of research." [1, p.28-29]

According to the Soviet Encyclopedic Dictionary, the term ***"paradigm"*** means an initial conceptual scheme, a model for posing problems and solving them, and research methods that are dominant during a certain historical period in the scientific community.

Below this word is used in the sense of the idea of ***the basic elements of programming and their interrelations***. The need to select from the entire diversity of programming ideas a certain minimum set of concepts that make up a paradigm is related to the fact that without their assimilation, successful independent work of the end user with the program is impossible. The latter is always important, but becomes especially relevant in the context of the widespread use of personal computers and the increasing involvement of specialists in problem areas in the use, as well as in the development of application systems.

In [2] it is rightly noted that "... the rules of a particular programming language can be learned in a few hours: the corresponding paradigms require much more time both to learn them and to unlearn them."

---

(1) Kuhn T. Structures of Scientific Revolutions. - M.: Progress, 1977. - 300 p.
(2) Fedyushin D. Programming Paradigms // INFO, 4, 1991, pp. 11-15; 5, 1991, pp. 13-17.

In the next step we will present ***a classification of paradigms***.

Step 45.
Classification of paradigms

In [1, p.195] the following classification of programming languages and styles is given:

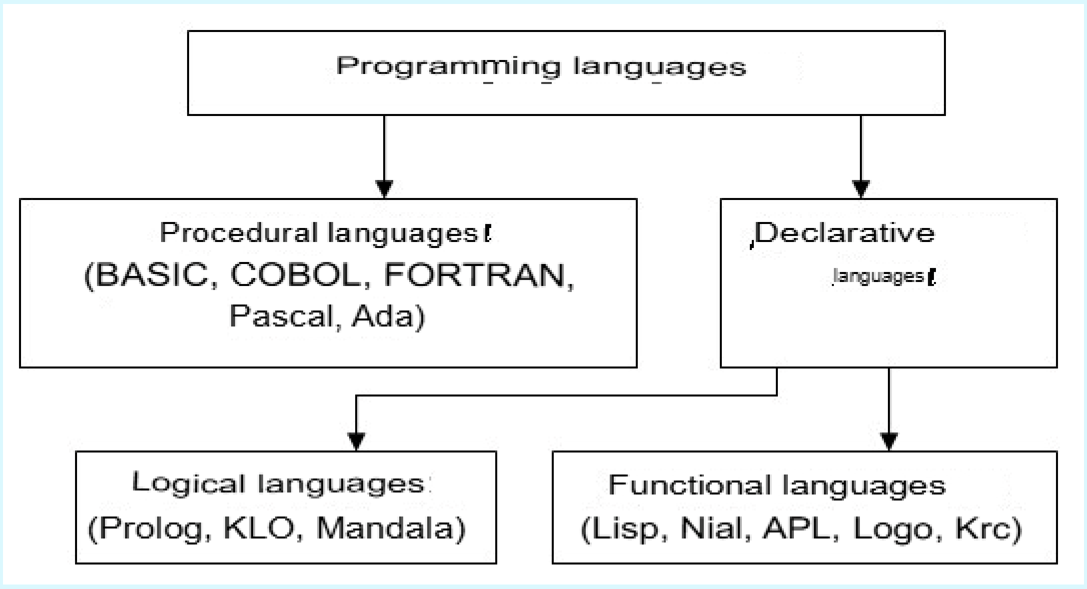
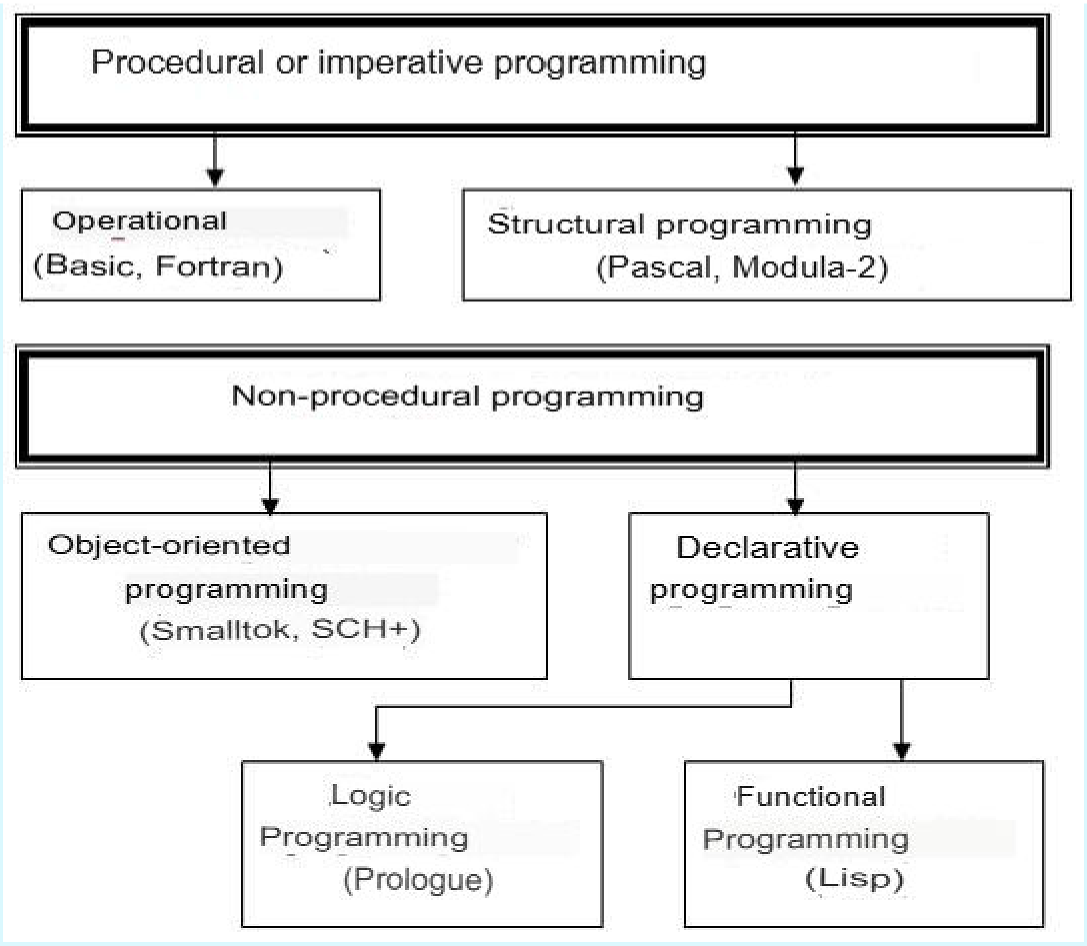

Programming languages
Procedural languages (BASIC, COBOL, FORTRAN, Pascal, Ada)
Declarative languages
Logical languages (Prolog, KLO, Mandala)
Functional languages (Lisp, Nial, APL, Logo, Krc)

Fig. 1. Classification of programming languages

In [2, p.11] we encounter a similar classification:



Procedural or imperative programming
Operational (Basic, Fortran)
Structural programming (Pascal, Modula-2)
Non-procedural programming
Object-oriented programming (Smalltok, SCH+)
Declarative programming
Logic Programming (Prologue)
Functional Programming (Lisp)

Fig.2. Classification of programming languages

Moreover, the author notes that the given classification does not claim to be complete; “there are other paradigms: for example, parallel, flow, production programming and smaller ones based on a specific method.”

We will try to expand and organize the given classifications. It is clear that any classification that satisfies the logical rules of division requires choosing ***the basis of division from the very beginning. We will choose the programming paradigm*** as the basis of division, or more precisely, the semantics (the model of the computational process).

It should be noted that when constructing classification systems, specialists in various fields of science increasingly use the principle of development, i.e. they base a classification on the development of higher forms of objects under study from lower ones [3, p. 140]. This allows us to prevent accidents and, in general, many errors in the classification of objects under study. However, such a method in itself, of course, cannot automatically lead to a correct classification.

First, let's look at explanatory dictionaries on computer science and computing [4-7].

A programming language **is** a formal language for representing programs or their parts within one or more programming systems. The number of “live”, i.e. currently used, programming languages is measured in hundreds. In connection with the executor, it is customary to distinguish ***machine languages*** and ***high-level*** programming languages. By the nature of semantics (i.e. the model of the computational process) in programming languages, two main moods can be distinguished - imperative (imperative, represented by operators, commands, instructions) and indicative (declarative, descriptive). In some programming languages, the description of actions, algorithms, i.e. the process that allows obtaining a result prevails. Such languages are calledimperative ***(*****FORTRAN, BASIC, ALGOL, PL / 1, PASCAL, C, ADA)**. Other programming languages \u200b\u200bare assumed not only the construction (calculation) of the result, but also the description (declaration) of its properties; on the basis of this information, the programming system itself must construct the algorithm. Such languages are called ***declarative***, non-procedural, problem-oriented; languages ***of relations,***

specifications, problem formulation, languages of artificial intelligence, automatic programming, algorithm synthesis.

The most significant classes of declarative languages are ***functional*** (or ***applicative***), ***production*** and ***logical languages***. The functional languages include, for example, **LISP, FP, APL, Nial, Krc** and **LOGO**, and the production languages include ***Refal***. The most famous logical programming language is **PROLOG**.

In practice, programming languages are not purely imperative, functional or logical (such languages are called ***metaphors*** (see [8, p. 23])), but contain features of languages of different paradigms. In a procedural language, one can often write a functional program or part of it, and vice versa.

Note that along with the term "***declarative*** programming" the term "***logical programming"*** can be used.

The fact is that the term ***"logic programming"***, which appeared around 1975, is interpreted in different ways.

In ***a narrow interpretation***, it is associated primarily with programming systems based on the use of special classes of logical formulas (***Horn clauses***) as logical programs and special methods of logical inference (variants of ***the resolution method***) as a logical model of computations or a method of executing logical programs. Therefore, ***logical programming in a narrow sense is sometimes called Horn, resolution or "prologue-like" programming***, although each of these epithets requires reservations, since it captures only part of the subject [9, p. 299].

In ***a broader interpretation***, logical programming includes a much larger range of concepts, methods, languages and systems, which is based on the idea of describing a problem as a set of statements in a certain ***formal logical language*** and obtaining a solution by constructing ***an inference*** in a certain formal (deductive) system [9, p. 299].

The classes of formulas used to describe problems, the methods for determining their semantics, the models of computations based on certain systems of derivation or transformation of formulas are very diverse. They form specific "styles", "types" or paradigms of programming that differ significantly from the traditional ones based on the models of computations embodied in traditional programming languages and in the architecture of typical computers of the first four generations (sometimes called ***von Neumann***). In addition to Horn (and/or resolution) programming, one can speak of ***equational programming, functional (applicative) programming*** and other "programmings", as well as their various combinations, united by the term "logical programming".

Thus, for now we can distinguish ***two paradigms at the lowest level*** of the programming paradigm tree:

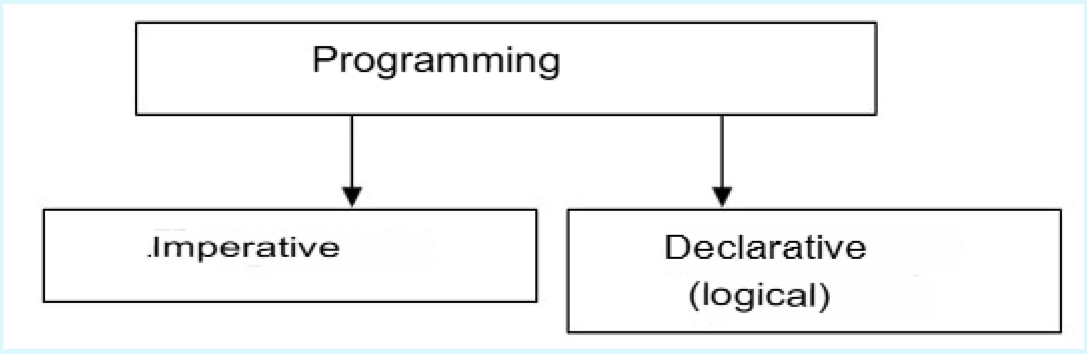


Fig.3. Lower-level paradigms

It is possible to indicate the prototypes of the indicated types of algorithmic languages in natural languages. For imperative languages, this is ***the imperative mood*** (imperative, command), for sentential languages - ***the indicative mood*** (description, narration). Turning to natural language, it is easy to notice that "the indicative mood is incomparably more widespread and forms, in essence, the basis of language, while the imperative mood is presented in the form of some special modification" [10]. Thus, it can be concluded that ***"the relative weight of the indicative mood is a measure of the development of language"*** [10].

Let's move on to considering ***the second level of programming paradigms***.

Let's consider the paradigm ***of programming that is not object-oriented***. This paradigm arose at the dawn of the development of computer technology, has a long history of development and, having been dominant for a long time, is gradually giving way to the object-oriented paradigm. It extends to all layers of software: from operating systems (virtual machine process management) to the application level. Computer science as a science developed precisely under its influence.

How is a paradigm created? It is formed at the stage of designing a computing environment, including high-level programming languages. The main factor taken into account when designing a computing environment is the ease and flexibility of mapping the subject area of the tasks being solved onto computing resources.

The paradigm of non-object-oriented programming is based on the idea of a machine being controlled by a program or a set of procedures (it is implied that the machine can be abstract, for example, **a Pascal** machine).

The characteristic relationship of elements in the paradigm is shown in the form of a diagram [11, p.15]:

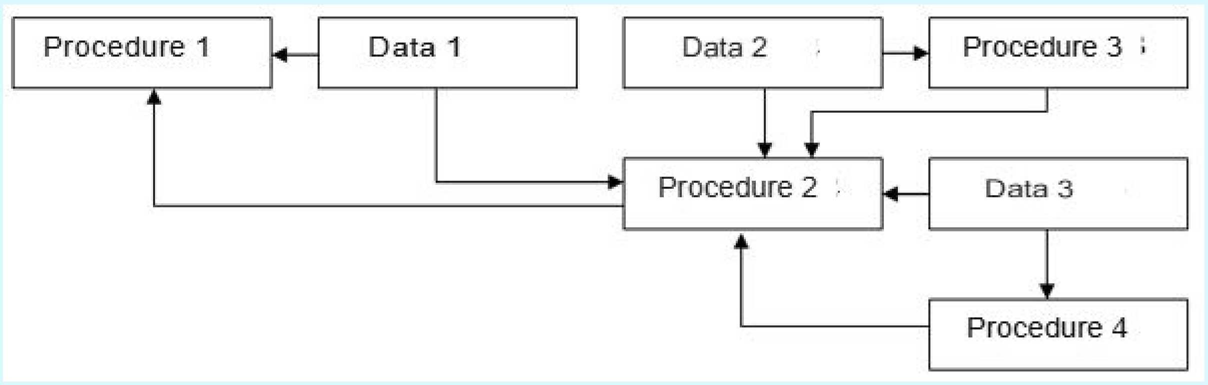


Fig.4. Interrelation of elements in the paradigm

The execution of procedures can naturally be called ***a process***, and one process can activate another, which is the main way of managing the overall process. At the moment of activation, data can be transferred, which in this case are called ***parameters***. The end of the procedure is associated with the return of the result and the control function to the calling program.

Let us pay attention to the following ***shortcomings*** inherent in the paradigm of non-object-oriented programming [11, p.15]:

1. The programming paradigm carries the space of processes, and the dominant ideas of people, which underlie our understanding of the world, are of an objective nature. The discrepancy between these ideas leads to difficulties in solving problems in problem areas on a computer.
2. There is an unnatural complexity of organizing the user's work in a process environment. For example, it is impossible to run a numerical task from a text editor or access a database to place the results in a document. The user must know many modes of operation in different processes and ways to achieve the desired process. Knowing the mysteries of moving from the current process to the desired process is an exorbitant price for many users to pay for solving the problems of their problem area. In this situation, the so-called "friendly" means of communicating with a computer placed in an unfriendly environment are of little help.
3. The paradigm greatly reduces the basic ability of data to model the real world. Even if data is global, rather than being passed as parameters, it still plays the role of procedure or function parameters in this paradigm. The data is said to be "dissolved" in the program.
4. It is impossible to create a mechanism for preserving the task context between interactive work sessions without involving additional representations. This mechanism represents the values of all data that "die" after the procedure is completed. Involving additional representations about the file system or database entails additional complexity in working with the computer.

5. The phenomenon of time does not find its expression in this paradigm. The change of time in our consciousness is connected with the change of states of some objects, for example, nature in spring, or one object, for example, the readings of your wristwatch. The beginning or end of the process of performing a procedure cannot serve as a moment of counting, therefore, without additional representations it is impossible to start a mechanism with a time counting function. The phenomenon of time is important for simulation tasks and for building real-time computing systems and operating systems.

From this we can conclude that a more suitable tool for formalizing knowledge on a computer would be a computing environment based on a different paradigm, based on a concept that is close to the concept of ***a system***, or a system model that has received general disciplinary distribution.

This formulation of the question leads to ***the paradigm of object-oriented programming***, which is based on the idea of the activity of data, not procedures, in which one can see the new paradigm's inversion in relation to the one considered above. Let us schematically depict the diagram of the new paradigm [11, p.15]:

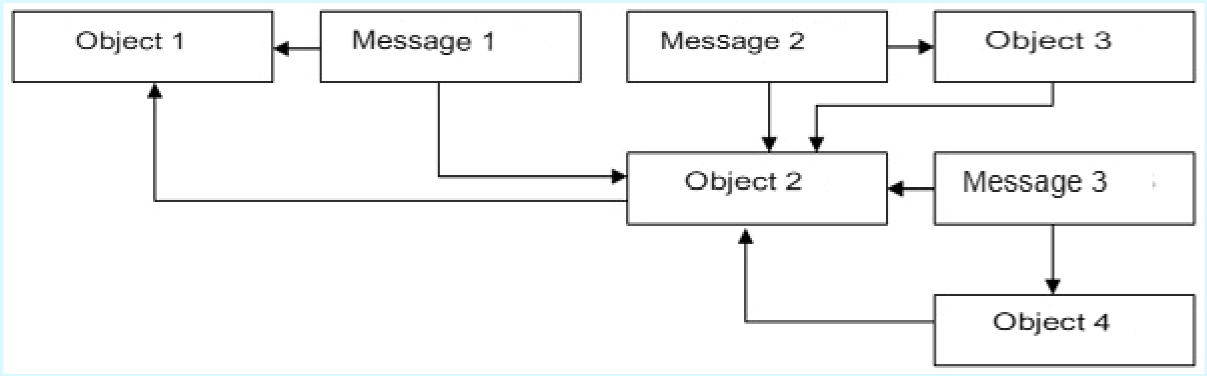


Fig. 5. Diagram of the object-oriented paradigm

At first glance, the diagrams above look identical, which might lead one to suspect that the two paradigms are not very different from each other. However, this is not the case. Suffice it to say that objects can change their value during program execution, and this reflects their nature as data. For a process within a non-object-oriented programming paradigm, self-modification of code is a bad programming style and is used extremely rarely.

The object-oriented programming paradigm includes three concepts:

- ***object***,
- ***message*** and
- ***method***.

Let us present a dictionary of key terms of object-oriented programming, borrowed from the monograph [103].

An object is a system component represented by private (own) memory and a set of operations.

A message is a request to an object to perform one of its operations.

A method is a description of how to perform one of an object's operations.

Class - a description of a group of similar objects.

An instance is one of the objects described by a class.

An object has the ability to interact with objects external to it. As such a means of interaction in object-oriented programming, the mechanism of sending and receiving ***messages*** is used, which, apparently, does not correspond well to the idea of interaction of systems, but nothing better has been invented.

So, in order to make an object perform some actions, i.e. start executing a program and get a result from it, it is necessary to send ***a request message*** to this object. In response to the request, the object itself can send a sequence of messages to other objects. In this case, the chain of messages is extended, and the message route can branch. Thus, the joint activity of objects within the computing system is organized.

A sequence of messages that will be sent by one object to a number of other objects, and perhaps to itself ***(recursion)***, as a reaction to receiving a message is ***a model of the object's behavior***. The object's reaction may be different depending on the type of message received.

Let us give, according to the article [11, p.16], a general characteristic of the object-oriented programming paradigm:

- the paradigm contains a natural picture of the space of objects, the interaction of which can be organized according to cause-and-effect relationships;
- since at any moment of work in an object-oriented environment the user has access to all global objects, he can easily move from one type of activity to another: editing text, viewing and analyzing data, performing calculations, processing images, etc. The transition is carried out in the same way;
- In contrast to the traditional paradigm, data is localized and structured naturally;
- preserving the context of a task between sessions of interactive work receives a natural expression: an object is not a process and cannot disappear;
- To keep track of time, a special object can be created - a timer, which can either send messages or respond to requests for time.

The presented analysis shows that the paradigm of object-oriented programming has advantages over the paradigm of traditional programming in formalizing knowledge, which is explained by the reduction of the so-called ***semantic gap*** - the gap between the principles of modeling real objects and the principles underlying programming languages. This gap in the work [11, p.16] is called ***the second semantic gap***, since G. Myers [13] has already introduced the concept of ***the semantic gap between programming languages and computer architecture***.

Procedurality is a specific part of our knowledge of the world, which is consistent with the concept of the special purpose of software for formalizing knowledge. Therefore, in order for software to be able to absorb knowledge, procedurality must be an integral part of programming languages.

But it seems that the object-oriented paradigm automatically excludes procedurality. No, this did not happen. In this regard, they say that ***procedures are encapsulated in an object***.

We are now able to complete the "paradigm tree", which remains binary:

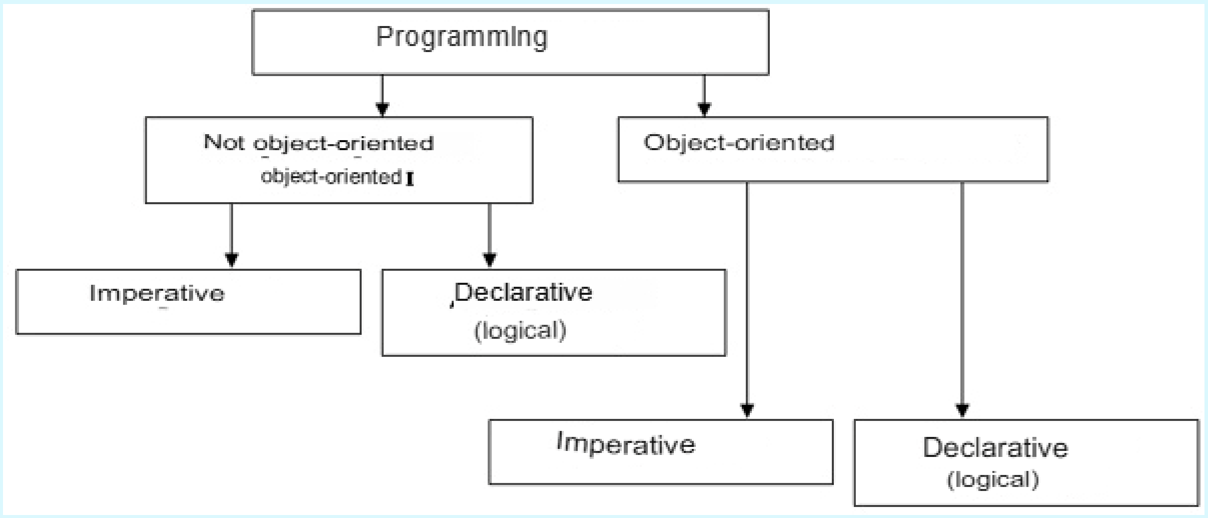


Fig.6. Paradigm tree

Finally, we will describe ***the third level of programming paradigms***, which contains programming paradigms that we have called ***metaparadigms***. These are ***sequential, parallel, competitive and distributed programming*** [14, p. 120-122]. These areas are distinguished in accordance with the means of interaction ***of processes***, methods of organizing processes and the purposes of their creation.

A process is a sequence of planned events determined by an object or phenomenon and carried out under given conditions; the course of events occurring in accordance with a planned goal or result [7].

Sequential programming is the well-known traditional programming. In it, attention is usually not focused on the concept of process [14, p.120].

Parallel programming is a set of language tools and methods for solving problems on a computer that allow parallel data processing. These can be multiprocessor systems with shared RAM, vector processors, associative processors, etc. The main area of application of parallel programming is solving complex computational problems. The main goal is to achieve maximum computer performance. A characteristic feature is that the generated processes usually have one program and operate on data of the same structure [14, pp. 120-121].

Concurrent programming (the term "concurrent" can be considered as a derivative of the word "competition") is programming in which a program solving a specific problem is represented as a set of multiple processes executed in parallel with each other. The programs for these processes are usually different. Here, the concept of parallelism is largely arbitrary, since if these processes are executed on a single computer, then at any given moment only one of them is actually executed by the processor. The illusion of parallelism is achieved by switching the processor from one process to another in accordance with a certain service discipline. What is important here is that these processes are executed on a single computer (possibly on a multiprocessor, but usually on a single-processor) and, as a result, compete with each other for its physical and logical resources: processor, input-output channels, memory areas, data sets, etc. [14, pp. 121-122].

Let us note two main milestones in the development of competitive programming.

1978. The article "Communicating Sequential Processes" by the British programmer and mathematician Charles Anthony Richard Hoare is published, laying the mathematical foundations for competitive programming.

1984. British **Semiconductor Manufacturer INMOS** develops the **Occam** language - an "assembler language" for computing systems built from many parallel-operating special microprocessors - transputers. The language is named after the medieval English scholastic philosopher and logician William of Occam (1285-1349) and is based on the mathematical theory of C.A.R. Hoare. The main concept of the language is ***a process***. Processes can be executed both sequentially and in parallel and interact using ***channels***.

Distributed *programming* is a set of language tools and methods for programming distributed data processing systems in computer networks and multi-machine complexes. It can be said that distributed programming is competitive programming without allowing processes to have common memory [14, p.119].

A distributed data processing system is a system whose individual components operate simultaneously on different computers that have the means to exchange data with each other [14, p.119].

Conclusion.

Programming paradigms have a hierarchical structure, represented as a tree, as shown in the diagrams below:

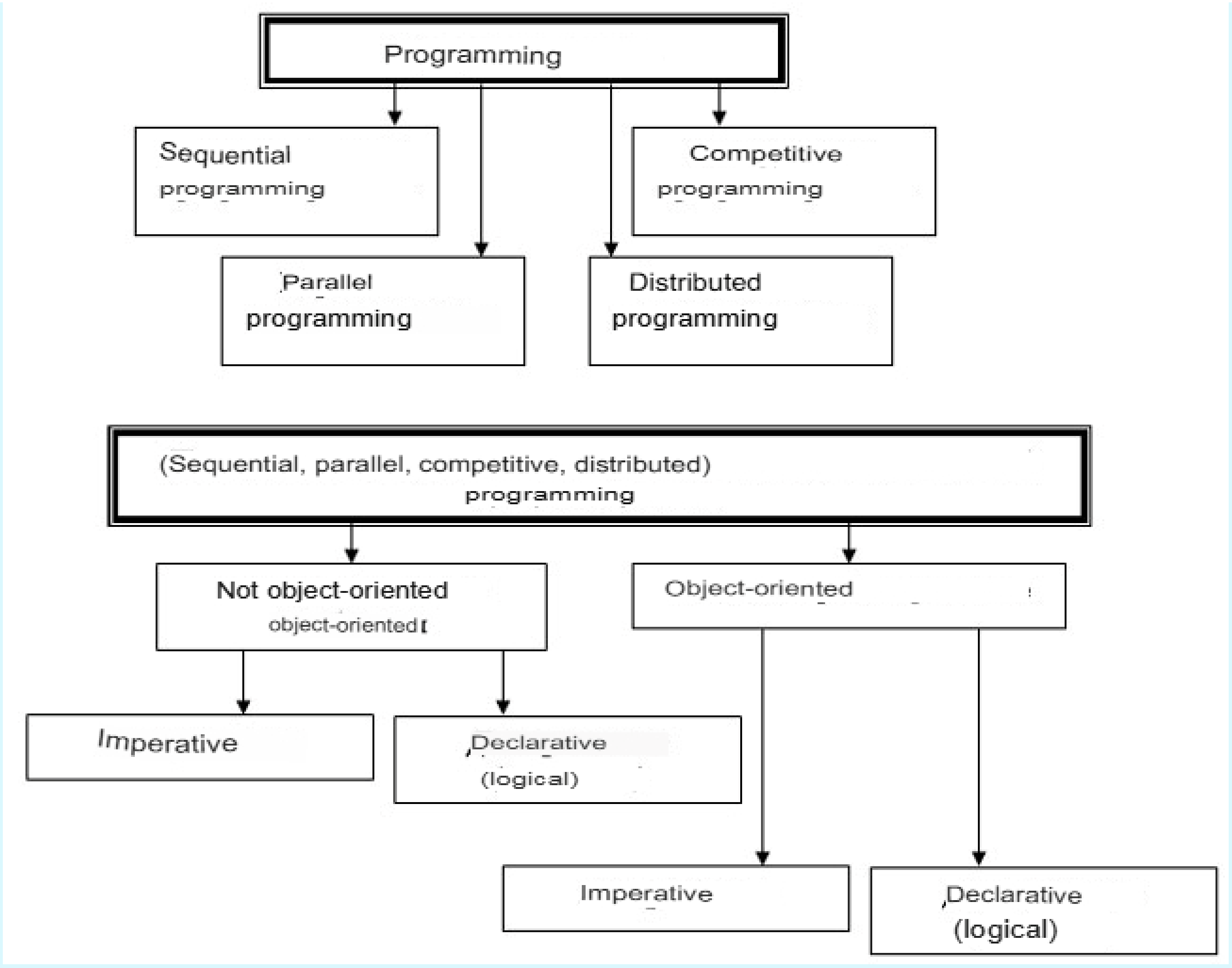


Fig. 7. Structure of programming paradigms

Thus, typical languages of competitive imperative programming that are not object-oriented are **Concurrent Pascal, Modula-2, Ada, Occam**.

---

(1). Hyvänen E., Seppänen J. The World of Lisp. In 2 volumes. Volume 1: Introduction to the Lisp Language and Functional Programming. - M.: Mir, 1990. - 447 p.
(2) Fedyushin D. Programming Paradigms // INFO, 4, 1991, pp. 11-15; 5, 1991, pp. 13-17.
(3) Formal Logic. - L.: Leningrad State University Publishing House, 1977. - 358 p.
(4) Borkovsky A.B. English-Russian Dictionary of Programming and Computer Science (with interpretations). - M.: Russian Language, 1989. - 335 p.
(5) Zamorin A.P., Markov A.S. Explanatory Dictionary of Computer Science and Programming: Basic Terms. - M.: Russkiy Yazyk, 1988. - 221 p.
(6) Explanatory Dictionary of Computing Systems / Ed. by V. Illingworth, E. L. Glazer, I. K. Pyle. - M.: Mashinostroenie, 1989. - 568 p.
(7) Pershikov V. I., Savinkov V. M. Explanatory Dictionary of Computer Science. - M.: Finance and Statistics, 1991. - 543 p.
(8) Hyvänen E., Seppänen J. The World of Lisp. In 2 volumes. Vol. 2: Programming Methods and Systems. - M.: Mir, 1990. - 319 p.
(9) Logical Programming. - M.: Mir, 1988. - 368 p.
(10) Basic Refal and its Implementation on Computing Machines. / TsNIPIASS. - M., 1977. - 238 p.
(11) Kryukov V.A. Analysis of principles of object-oriented programming // Microprocessor tools and systems, 1989, 2, pp.14-22.
(12) Goldberg A. and Robson D. SmallTalk-80: The Language and Its Implementation. - Addison-Wesley, Reading, Mass. 1983.

[13] Myers G. Architecture of modern computers: In 2 books. Book 1. - M.: Mir, 1985. - 384 p.
[14] Dedkov A.F. Abstract data types in the AT-Pascal language. - M.: Nauka, 1989. - 200 p.

In the next step we will take a closer look at ***the object-oriented programming paradigm***.

Step 46.
OOP as a technological paradigm

In this step we will look at ***the object-oriented paradigm*** in more detail.

The article [1] lists the main directions of object-oriented programming and the programming languages that support each direction:

Table 1. **Directions of the OOP**

Direction...	**message oriented**	**operations oriented**
class/type oriented	**Smalltalk, C++ virtuals**	**CLOS, Fortran/C++ overloads**
object oriented	**Self, Actors**	**Prolog, Data-driven**

There are different opinions about whether the method of object formation is the main issue in object-oriented programming, which puts ***the object*** at the forefront. When considering object-oriented languages from the standpoint of constructing computational models, it can also be considered that ***the sending of messages*** as a mechanism for organizing the computational process is an essential point [2, pp. 87-88].

Our opinion on this issue is that ***object-centric OOP is a programming technology, and message-centric OOP is a programming paradigm***.

In this step we will look at object-oriented programming as ***a technology***.

First, we need to discuss the question of who needs programming technology and why.

In the monograph [3, p.10] the target problem is posed: “All that professional programmers can do to solve the central problem of information technology of the 80s - the formalization of knowledge - is to try to create ***a standard technology*** (or a range of standard technological methods, for example, in the main problem areas) for ***the automatic formalization of knowledge***, i.e. to develop tools that make it easier for non-programming professionals to independently formalize their individual knowledge.”

Programming technology is needed:

- to the head of a large software development project;
- ***one programmer*** making a large program in order to keep the project in mind as a whole, clarifying details if necessary.

Looking ahead a little, we note that object-oriented programming is a direct consequence of the increasing complexity of modern applications; for example, inheritance and encapsulation are the most effective means for combating the increasing complexity of software projects.

Like any technology, programming technology is not aimed at raising the level of the best programmers to an unattainable height, but, on the contrary, at ensuring that the bulk of programmers can produce good programs.

And the development of each element of technology (such as, for example, structured programming or OOP) goes through three stages - creative, ideological and instrumental.

At ***the creative stage,*** outstanding programmers invent programming rules for their own use and apply them successfully. This was the case, for example, with structured programming in the 1960s. Then other people (usually they are not even programmers) formalize these rules as ***an ideology***. And finally, ***a set of tools*** is created for the convenient application of this ideology, and newcomers do not always know and understand the ideology. For them, the entire technology consists of these tools. In this way, average-level programmers and even beginners can produce technologically advanced programs. (And high-level programmers either become heretics, criticizing these tools, and sometimes the ideology itself, or develop a new one.)

We distinguish three technological paradigms of programming:

- ***structured programming***,
- ***programming using abstract data types (ATD programming)*** and
- ***object-oriented programming***.

Structured ***programming*** **is a** programming method that involves the creation of understandable, locally simple and easy-to-read programs, the characteristic features of which are modularity, the use of unified structures of sequence, selection and repetition, the rejection of unstructured control transfers, and limited use of global variables [4-7].

Abstract data type programming and object-oriented programming are new approaches to structuring programs aimed at improving the quality of programs (reliability, modifiability, knowability).

Abstract data type programming languages (ADT languages) are programming languages that support the technology of designing programs using abstract data types defined by the programmer. Some of the first languages of this type are **CLU, Alphard,** and **Ada**.

Abstract ***Data Type*** **-** a data type defined only by operations applicable to objects of a given type, without describing the method of representing their values [4-7].

Object **- oriented *programming languages*** are programming languages in which a program is defined by describing the behavior of a set of interconnected ***objects***.

Objects exchange ***requests***; in response to a received request, an object sends requests to other objects, receives responses, changes the values of its internal variables, and returns a response to the received request. The request mechanism in object-oriented languages is interesting in that when an object executes a request, only the values of the variables of this object can be directly changed. Unlike a procedure, which describes how processing should be performed, a request only defines what the sender wants to do, and the recipient defines exactly what should happen. The object-oriented programming paradigm introduces modularity into a program through data abstraction and inheritance and is especially well suited for situations where there is a clear hierarchical classification of objects. This allows one to avoid duplication and localize descriptions of the mechanisms for working with information. One of the first languages of this type was **Smalltalk** [4-7].

Like any technology, object-oriented programming is a discipline that you must "impose" on yourself using the tools provided by the language.

Note that the concept of an object is based on structured programming methods and data abstraction-based software development methods.

Structured programming is associated with functional decomposition and involves designing a software product "from top to bottom". However, this method does not allow for the dependence of the program architecture on the data structures that it will have to process.

Using an approach based on data abstraction leads to the opposite effect: the program is developed "from the data", and the emphasis is on choosing a method for their presentation. In this case, naturally, a gap is formed between the data structures and the procedures for their processing.

Object-oriented programming eliminates the opposition of procedures to data and their inequality, which are characteristic of the two approaches mentioned, and at the same time integrates the advantages of the considered methods of program development.

Thus, object-oriented programming supports a qualitatively new level of ***joint*** structuring of data and procedures for their processing.

According to Alex Lane (PC World, 1991, 5, p.35), for the way programs are written to truly change, object-oriented programming systems must ***grow out of traditional languages rather than displace them***. Backward compatibility - both with previously written programs and with accumulated programming experience - is of decisive importance.

C++ is a good example of such an evolutionary approach. In addition, object-oriented codes should be generated in such a way as to allow their subsequent optimization and to give the programmer the ability to control the maximum number of parameters. Features such as machine code generation, the admissibility of traditional methods and objects, constructors and destructors are not a luxury, but a necessity.

Object-oriented programming has ***its own set of concepts***.

The basis of OOP is the concept of ***"object"*, similar to the Pascal** "record" data type (or "structure" in **C**), but with one significant difference: objects include not only data, but also procedures and functions called ***methods***. The ability to combine data and rules into one type allows you to develop programs at a logical level.

An object type is a structure consisting of a fixed number of components. Each component is either a field containing data of a strictly defined type, or a method that performs operations on the object. By analogy with the declaration of variables, a field declaration specifies the data type of the field and an identifier naming the field: by analogy with the declaration of a procedure or function, a method declaration specifies the title of a procedure, function, constructor, or garbage collector.

In addition, object-oriented programming, which we consider as ***a technology***, is based on three more basic concepts: ***encapsulation, inheritance and polymorphism***.

The main methodological principle of OOP is that objects model the characteristics and behavior of elements of the world in which we live.

1. The combination of code and data in an object is called ***encapsulation***.

   As we have already said, combining records with procedures and functions that manipulate the fields of these records forms a new data type - ***an object***.

   How do you access an object's fields? How do you assign values to them? An object's data fields are what the object knows, and the object's methods are what the object does. To access the object's data fields, you must use ***the object's methods***.

   A method is a procedure or function declared within an object and tightly scoped to that object.

   One of the most important principles of object-oriented programming is that the programmer ***must think about code and data together*** when developing a program. Neither code nor data exists in a vacuum. Data controls the flow of code, and code manipulates the images and values of data. If your code and data are ***separate entities***, there is always the danger of calling the right procedure with the wrong data, or the wrong procedure with the right data. It is the programmer's responsibility to ensure that these entities coincide.

An object synchronizes code and data by constructing their declarations together. In fact, to get the value of one of an object's fields, you call a method on that object that returns the value of the field. To assign a value to a field, you call a method that assigns a new value to that field.

Some object-oriented languages, such as **Smalltalk**, require encapsulation, but in languages such as **Turbo Pascal** you have a choice.

Equally important is the fact that objects can inherit characteristics and behavior from what we call ***their parent objects*** (or ***ancestors***).

2. The goal of science is to describe the interactions of the universe. Much of the work in science, in pursuing this goal, is simply constructing family trees. When an entomologist returns from the Amazon with a previously unknown insect in a jar, his main concern is to determine where it fits into the giant chart that lists the scientific names of all the other insects. Similar charts have been drawn for plants, fish, mammals, reptiles, chemical elements, elementary particles, and galaxies. They all look like family trees: with a single, overarching category at the top and an ever-increasing number of categories below them.

This process of classification is called ***taxonomy***. It is a great starting metaphor for the inheritance mechanism in object-oriented programming.

The questions a scientist asks when trying to classify some animal or object are:

- how this object is similar to other objects from the same class;
- how it differs from other objects.

Each particular class has a set of "behaviors" and characteristics that define that class. The scientist starts at the top of a particular family tree and works his way down the child regions, asking himself these two questions. The highest level is the most general, and the questions are the simplest, such as winged or wingless? Each successive level is more specific than the previous one, and less general. Eventually, the scientist gets to the point of counting the hairs on the third segment of the insect's hind leg!

The important thing to remember is that once a characteristic is defined, all categories below that definition contain that characteristic. So once you have identified an insect as a member of the order **diptera** (flies), you do not need to note again that flies have one pair of wings. The species of insect we call flies inherits that characteristic from their order.

In this way, we note the ability of objects ***to inherit*** data and rules from other objects. The process by which one type inherits the characteristics of another type is called ***inheritance***.

The inheritor is called ***the derived type***, and the type that the child type inherits from is called ***the parent type***. Inheritance is ***transitive***, i.e. if **T3** inherits from **T2**, and **T2** inherits from **T1**, then **T3** inherits from **T1**. The ownership of an object type consists of itself and all its descendants.

For example, let's say we have a graphic object **Circle**. This is a circle of a given radius, displayed in a certain color in a certain place on the display screen. Using the inheritance mechanism, we can create a **FilledCircle** object by defining only one new element - **the method** by which the circle will be filled with the desired color.

Thus, inheritance allows you to reduce the size of the code to a minimum and reuse the already created code.

If a derived type is defined, the methods of the derived type are inherited, but they can be ***suppressed*** if desired. For example, to suppress an inherited method, simply declare a new method with the same name as the inherited method, but with a different body and (optionally) a different set of parameters.

As you can see, object-oriented programming is largely ***a process of building a family tree for data structures***. One of the important features that object-oriented programming adds to imperative languages like **Pascal** is a mechanism by which data types can inherit characteristics from simpler, more general types.

3. When objects are organized hierarchically, they can contain rules with the same name and different actions. For example, **Circle** and **Box** objects can include different rules for displaying them on the screen. Let **Circle** and **Box** objects be descendants of **Shape**. Then all the program needs to know about the object is that it is of type **Shape**. When the "display" rule is called, the specifics of displaying a particular shape (**Circle** or **Box**) will be taken into account by the object itself.

Polymorphism is the assignment of a single name to an action, which is then shared "up" and "down" through a hierarchy of objects, with each object in the hierarchy performing the action in a way that is appropriate to it.

In other words, ***polymorphism*** is the ability of objects to follow different rules of their own, even though they have the same name.

Object-oriented programming fanatically "animates" objects because the randomness and regularity that fills our lives have characteristics (data) and behavior patterns (methods).

For example, the characteristics of a toaster may include the voltage it requires, the number of pieces of toast it can toast at one time, the low or high toasting setting, the color of the toaster, its brand name, etc. Its behavior may include loading pieces of bread, toasting those pieces, and automatically ejecting the toasted pieces.

If we want to write a program to simulate a kitchen, what is the best way to model the various devices other than objects, with their characteristics and behaviors encoded in data fields and methods? In fact, this has already been done: one of the first object-oriented languages (**SIMULA-67**) was created as a language for writing such simulations.

From now on, data are not containers for you that you can fill with values! From the point of view of the new view of things, objects look like actors on a stage with many memorized roles (methods). If you (the conductor) give them the floor, the actors begin to recite according to the script.

For example, in **Turbo Pascal,** the operator **APoint.MoveTo(242,118)** can be "animated" like this: this is an instruction to the **APoint** object to "Move itself to position 242,118."

There is also a reason why OOP is tied quite tightly in the traditional sense to ***a graphics-*** oriented environment. Objects are supposed to be models, and what better way to model an object than to draw a picture of it?

Let us present the main milestones in the development of object-oriented programming [2, pp. 88-90].

Second half of the 60s. W. Dahl and his colleagues (Norway) developed the **SIMULA-67** modeling language, which embodied the initial ideas of object-oriented programming. Since **SIMULA-67** is a modeling language, it requires descriptions of objects and their actions. Here, there are already descriptions called ***"class"* (class)**, and constructions are allowed in which inheritance of objects is used.

1983. **Smalltalk**, the first object-oriented programming language, was released onto the market and became widely known. Ithad been developed since the early 1970s ***by Alan Kay*** and a group of his colleagues at the Xerox Research Center in Palo Alto (USA). It implemented computation mechanisms by describing the contents of objects, describing actions, inheritance, and passing messages.

1983. A group of researchers headed by ***B. Stroustrup*** completed the development of the **C++** programming language, which is the result of further development of the **C** programming language in the direction of object orientation and inclusion of type control mechanisms, data abstraction and operation combination. The name **C++** was invented by Rick Massitti. The name indicates the evolutionary nature of the transition to it from **C.** "++" is the increment operation in C. B. Stroustrup [8, p.10] jokes that “experts in language semantics find that **C++** is worse than **++C**”.

Conclusion.

Object-oriented programming should be in the 1990s what structured programming was in the 1980s - the most effective means of dealing with the complexity of software systems.

---

(1) Gabriel RP, White JL, Bobrow DG CLOS: Integrating Object-Oriented and Functional Programming // Communications of the ACM, v.34,N9 (September 1991), pp.28-38.
(2) Futi K., Suzuki N. Programming languages and circuit design of SBIS. - Moscow: Mir, 1988. - 224 p.
(3) Gromov GR National information resources: problems of industrial operation. - Moscow: Nauka, 1984. - 237 p.
(4) Borkovsky AB English-Russian dictionary of programming and computer science (with interpretations). - Moscow: Russian language, 1989. - 335 p.
(5) Zamorin AP, Markov AS Explanatory dictionary of computing technology and programming: Basic terms. - M.: Russian language, 1988. - 221 p.
(6) Explanatory dictionary of computing systems / Ed. by V. Illingworth, E. L. Glazer, I. K. Pyle. - M.: Mechanical engineering, 1989. - 568 p.
(7) Pershikov V. I., Savinkov V. M. Explanatory dictionary of computer science. - M.: Finance and statistics, 1991. - 543 p.
(8) Stroustrup B. The C++ programming language. - M.: Radio and communication, 1991. - 352 p.

In the next step we will look at ***the imperative paradigm*** in more detail.

Step 47.
The imperative paradigm

In this step we will look in more detail at the essence of ***the imperative paradigm***.

As before, let us first agree on terminology [1-4].

Imperative Languages **(Imperative (English) -** imperative, containing an indication to perform some action) - a class of programming or database manipulation languages. A program written in an imperative language explicitly specifies the method of obtaining the desired result, without defining the expected properties of the result - the result is implicitly specified only by the method of obtaining it using a certain procedure. The procedure for obtaining the desired result is in the form of a sequence of operations, therefore, procedural languages are characterized by the indication of the control logic in the program and the order of execution of operators. Typical operators are assignment and control transfer operators, input-output operators and special operators for organizing cycles. They can be used to create program fragments and subroutines. This type of programming is based on taking the value of some variable, performing an action on it and saving the new value using the assignment operator, and so on until the desired final value is obtained. Note that assignment operators are information-destructive (the assigned value replaces the previous value of the variable) and are also order-dependent. Imperative languages are closely related to the von Neumann model of computation, and therefore most popular languages are imperative.

A procedure- oriented *language* is an imperative programming language based on the concept of ***a procedure and a variable***. A procedure performs some action using and changing the values of variables that are its parameters, as well as global and local variables.

The action of a procedure is described by a sequence of simpler actions performed by calling other procedures and basic language operators. Examples of procedural programming languages are **FORTRAN, ALGOL-60, PL/1, COBOL, Pascal, Ada,** etc.

Note that the following interpretation of the term "procedural language" is very common: a procedural language is the same as an imperative programming language. The term "procedural language" should be considered unsuccessful, because in programming a procedure is associated with a more special concept than that implied by the definition "procedural" in a procedural language.

Modular programming language **(MPL)** is a programming language in which a program is organized as a set of ***modules*** with strict adherence to the rules of their interaction; the description of a module consists of a description of the interface and a description of the implementation. In the simplest case, a module is a procedure; modern languages have more developed means of modularity: ***packages*** and ***tasks* in the Ada** language, ***modules*** in the **Modula-2** language, ***abstract data types***.

All of the above can be represented as a tree:

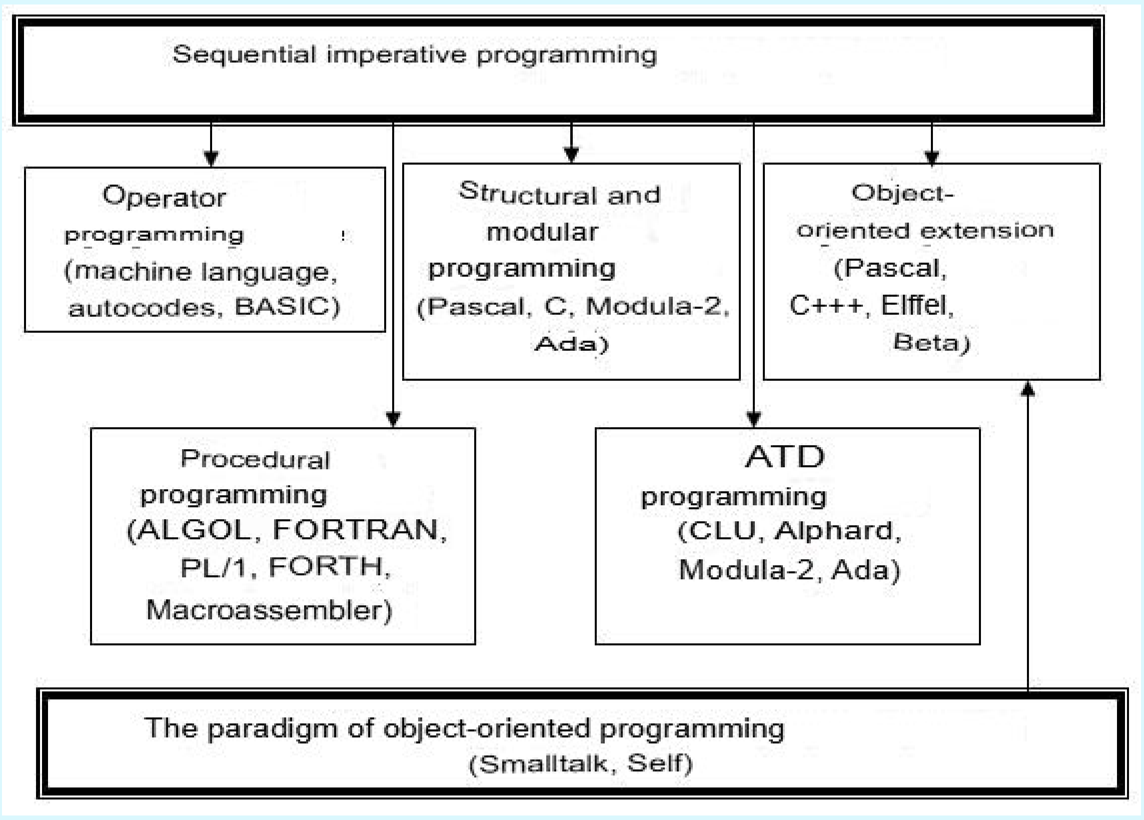

Sequential imperative programming
Operator programming (machine language, autocodes, BASIC)
Structural and modular programming (Pascal, C, Modula-2, Ada)
Object-oriented extension (Pascal, C++, Elffel, Beta)
Procedural programming (ALGOL, FORTRAN, PL/1, FORTH, Macroassembler)
ATD programming (CLU, Alphard, Modula-2, Ada)
The paradigm of object-oriented programming (Smalltalk, Self)

Fig. 1. Imperative programming tree

The history of the development of imperative programming is rich in events:

1936: Twenty-five-year-old Cambridge University student Englishman ***Alan Turing*** published an article "On Computable Numbers", in which he considered a hypothetical device (the "Turing machine") suitable for solving any solvable mathematical or logical problem - the prototype of a programmable computer.

1945: American mathematician ***John von Neumann's*** report "Preliminary Report on the Advac Machine" is sent out, containing the concept of storing computer commands in its own internal memory.

1972: Dennis Ritchie, a 31-year-old systems programmer at **Bell Labs,** developed the **C** programming language.

1973: Swiss programming expert ***Niklaus Wirth*** published the "Revised Message", which defined the exact standard for the **Pascal** language.

1978: Edsger Dijkstra's book “The Discipline of Programming” is published, devoted to fundamental issues of program construction and presenting a fundamentally new point of view on the essence of programming.

1979: The winner of the competition to create a programming language for the US Department of Defense's computer systems is announced. It is the Ada language, developed by **CII Honeywell Bull** under the leadership of Frenchman ***Jean Ichbia***. ***1981***. The textbook "The Science of Programming" by Cornell University (USA) professor ***David Gries*** is published.

A compiler for the programming language **Modula-2**, developed ***by Niklaus Wirth*** and the successor of the **Pascal** and **Modula** languages, has been handed over to users. The main features of the **Modula-2** language are the implementation of programs as a set of independent modules and multiprogramming.

1983: A group of researchers headed by ***B. Stroustrup*** completed the development of the **C++** programming language, which is the result of further development of the **C** programming language in the direction of object orientation and the inclusion of type control mechanisms, data abstraction and operation combination. The name **C++** was invented by Rick Massitti. The name indicates the evolutionary nature of the transition to it from **C.** "++" is the increment operation in C. B. Stroustrup [5, p.10] jokes that "experts in language semantics find C++ worse than ++C".

---

(1) Borkovsky A.B. English-Russian Dictionary of Programming and Computer Science (with Explanations). - M.: Russian Language, 1989. - 335 p.
(2) Zamorin A.P., Markov A.S. Explanatory Dictionary of Computer Science and Programming: Basic Terms. - M.: Russian Language, 1988. - 221 p.
(3) Explanatory Dictionary of Computing Systems / Ed. by V. Illingworth, E.L. Glazer, I.K. Pyle. - M.: Mechanical Engineering, 1989. - 568 p.
(4) Pershikov V.I., Savinkov V.M. Explanatory Dictionary of Computer Science. - M.: Finance and Statistics, 1991. - 543 p.
(5) Stroustrup B. The C++ Programming Language. - M.: Radio and Communications, 1991. - 352 p.

In the next step we will consider ***the declarative (logical) paradigm***.

Step 48.
Declarative (logical) paradigm

In this step we will dwell in more detail on ***the declarative (logical) paradigm***.

In explanatory dictionaries [1-4] you can read that **declarative *languages*** are a class of programming languages, the program in which specifies connections and relationships between objects and quantities and does not determine the sequence of actions. When using a declarative language, the program explicitly specifies what properties the result should have, but does not say how it will be obtained; any method of obtaining a result with the required properties is suitable.

Because declarative languages are based on static rather than dynamic concepts (i.e., they define "what" rather than "how"), concepts such as ordering and control flow are irrelevant to these languages, and they do not contain any assignment statements. Ideally, a program in a declarative language would consist only of an unordered system of equations characterizing the desired result. However, since a real program must be easy to implement and reasonably efficient, such an ideal model does not hold - for this, existing declarative languages would have to have absolutely perfect syntax and style. Declarative languages are not tied to the classical von Neumann model of computation, and in a typical case the algorithm for achieving the desired result may have a high degree of parallelism. The degree to which a language is "declarative" is a relative concept: **PROLOG** is declarative compared to assembly language, but it can be considered imperative compared to ***knowledge representation languages***.

J. Robinson writes [5, p.13]: "Functional programming appears to be a concept independent and somewhat separate from logical programming. However, I argue that both are examples of a more general, fundamental, unified, and simple idea that could be called... "assertive programming." This is the type of programming where you assert that some propositions are true, and then ask to derive other propositions as their consequences.

In logical programming these asserted propositions have the form of implications, and in functional programming they have the form of equalities (equations). But this is really only an external difference. I think ***the main thing here is that when you run systems of one kind or another, you run deductive machines: you ask them to perform the necessary deductions (conclusions) for you.*** "

Let us present the "architecture" of the declarative (logical) paradigm, which, in our opinion, is a useful tool for forming a view on the methodology of teaching programming:

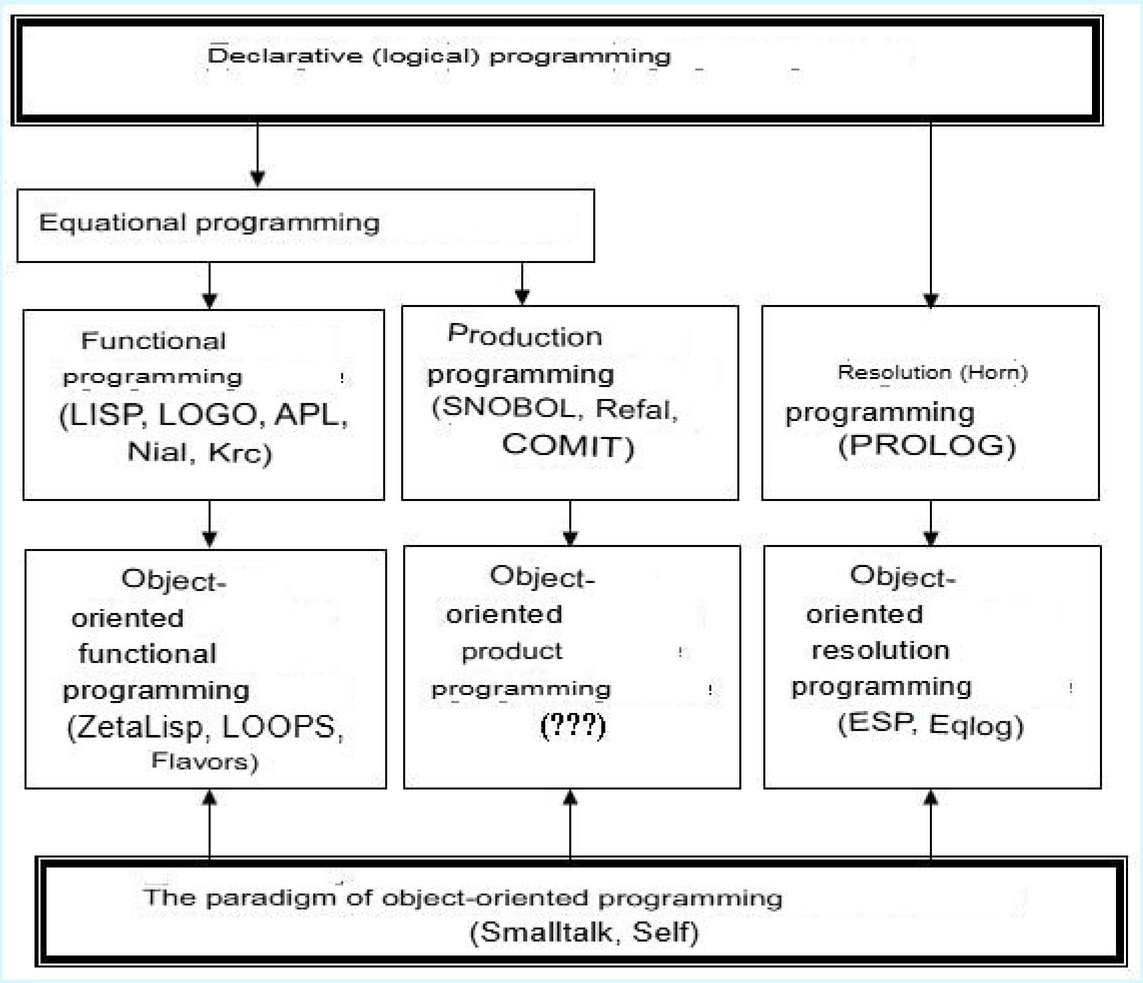

Declarative (logical) programming
Equational programming
Functional programming (LISP, LOGO, APL, Nial, Krc)
Production programming (SNOBOL, Refal, COMIT)
Resolution (Horn) programming (PROLOG)
Object-oriented functional programming (ZetaLisp, LOOPS, Flavors)
Object-oriented product programming (???)
Object-oriented resolution programming (ESP, Eqlog)
The paradigm of object-oriented programming (Smalltalk, Self)

Fig. 1. Tree of the logical paradigm

The "???" symbols in the diagram mean that we are not aware of any object-oriented implementations of production programming languages.

We will classify languages whose semantics are based on ***substitution systems*** **as equational programming languages. The languages OBJ, AFFIRM, and MAPC** are often included in this paradigm. Since these are experimental languages that do not claim high implementation efficiency, it is perhaps more appropriate to call ***them equational specification languages***.

---

(1) Borkovsky A.B. English-Russian Dictionary of Programming and Computer Science (with Explanations). - M.: Russian Language, 1989. - 335 p.
(2) Zamorin A.P., Markov A.S. Explanatory Dictionary of Computer Science and Programming: Basic Terms. - M.: Russian Language, 1988. - 221 p.
(3) Explanatory Dictionary of Computing Systems / Ed. by V. Illingworth, E.L. Glazer, I.K. Pyle. - M.: Mechanical Engineering, 1989. - 568 p.
(4) Pershikov V.I., Savinkov V.M. Explanatory Dictionary of Computer Science. - M.: Finance and Statistics, 1991. - 543 p.
(5) Logical programming. - M.: Mir, 1988. - 368 p.

From the next step we will begin to consider the paradigms listed in Figure 1, in particular, ***the resolution (Horn) paradigm***.

Step 49.
Resolution (Khornov) paradigm

In this step we will take a closer look at ***the resolution (Horne) paradigm***.

In explanatory dictionaries [1-4] you will read the following...

Rule-Oriented Programming *Languages* (RPLs) are a class of programming languages and a subclass of declarative languages that are based on symbolic logic. The logical paradigm is based on the use of a theorem-proving mechanism that allows one to find out whether a certain set of logical formulas is inconsistent. In this case, a program can be considered as a set of logical formulas together with a theorem (query) that must be proven. Formulas are intended to represent various facts (data) and inference rules. Databases and pattern matching (***unification***) are used to store and efficiently work with facts and rules. In this case, the programmer is relieved of the need to determine the exact sequence of steps for performing calculations. Logical programming languages are important due to their declarative nature, wide potential capabilities, flexibility, and suitability for implementation within the framework of highly ***parallel*** computing architectures. A typical representative of this paradigm is the **PROLOG** language, which is based on a subset of Horn expressions, but contains some "added" elements to bring it closer to natural language.

The Horn program consists of a set of sentences in the language of first-order predicate logic, which are often called ***facts and rules***. The structure of these sentences is such that, on the one hand, many problems are described quite naturally with their help, and on the other hand, they allow a simple procedural interpretation. These two circumstances allow the language of facts and rules to be used as a programming language. As has already been said, the idea of logical programming is quite neutral both in relation to the class of formulas used and in relation to the method of finding an inference, however, as practical experience has shown, not all procedures for constructing an inference are effective.

It is hardly possible to follow the entire history of Horn programming within the framework of a small section (especially since it is perfectly reflected in general terms in the lecture of the creator of the resolution method, J. Robinson [5], translated into Russian). We will tell only about the most basic facts from ***the history of the development of the Horn paradigm***.

1965. American ***J. Robinson*** publishes the article "Machine-oriented logic based on the resolution principle" (there is a translation of this article in the Cybernetic Collection, issue 7, 1970, pp. 194-218), containing the mathematical foundations of resolution programming.

1972. Frenchman ***Alain Colmerauer*** from the University of Lumini (Marseille) developed the **PROLOG** language.

---

(1) Borkovsky A.B. English-Russian Dictionary of Programming and Computer Science (with Explanations). - M.: Russian Language, 1989. - 335 p.
(2) Zamorin A.P., Markov A.S. Explanatory Dictionary of Computer Science and Programming: Basic Terms. - M.: Russian Language, 1988. - 221 p.
(3) Explanatory Dictionary of Computing Systems / Ed. by V. Illingworth, E.L. Glazer, I.K. Pyle. - M.: Mechanical Engineering, 1989. - 568 p.
(4) Pershikov V.I., Savinkov V.M. Explanatory Dictionary of Computer Science. - M.: Finance and Statistics, 1991. - 543 p.
(5) Logical programming. - M.: Mir, 1988. - 368 p.

In the next step we will consider ***the equational paradigm***.

Step 50.
Equational paradigm

In this step we will take a closer look at ***the equational paradigm***.

Equalities are ***formulas*** of a special kind

```
                    p = q
```

where **p** and **q** are ***expressions (terms)*** constructed from symbols (names) of functions, variables and constants.

Logic in which the formulas are only ***equalities*** is called ***equational logic*** or ***the logic of equalities***, and programming using equalities alone is called ***equational programming*** [1, p.325].

Equalities are a convenient and natural apparatus used to define functions, relations, sets, etc. However, it should be borne in mind that their ***semantics*** (mathematical meaning) can be defined in different ways. Some definitions can be more ***operational***, ***procedural***, while others are more ***declarative, non-procedural***.

In the context of equational programming, we would like to draw your attention to two variants ***of the operational*** definition that allow us to consider an equality or a system of equalities as a "logical program" defining "logical computations" that are not similar to computations in traditional programming languages or in machines of traditional architecture.

---

(1) Logical programming. - M.: Mir, 1988. - 368 p.

In the next step we will focus on ***the functional paradigm***.

Step 51.
Functional paradigm

In this step we will look at ***the emergence and development of the functional paradigm***.

Functional programming languages **(Functional Languages, Applicative Languages)** are a class of programming languages and a subclass of declarative languages based on the ideas of lambda calculus and recursive function theory. They are based on the concept ***of a function*** - a description of the dependence of a result on arguments using other functions and elementary operations. In functional languages, there is no concept of a variable and assignment, so the value of a function depends only on its arguments and does not depend on the order of calculations. A functional program consists of a set of function definitions. Functions, in turn, are calls to other functions and clauses that control the sequence of calls. Calculations begin with a call to a function, which in turn calls the functions included in its definition, etc. in accordance with the hierarchy of definitions and the structure of conditional clauses. Functions often call themselves either directly or indirectly. Each call returns a value to the function that called it, the calculation of which then continues; This process is repeated until the function that started the calculations returns the final result to the user. "Pure" functional programming does not recognize assignments and control transfers. Branching of calculations is based on the mechanism for processing the arguments of a conditional statement. Repeated calculations are carried out by means of ***recursion***, which is the main tool of functional programming. The most famous representative of languages based on the functional paradigm is the **LISP** language.

Modern **LISP** systems, as a rule, include one or several means of ***object-oriented programming***: **Flavors, Common Loops** (for the **Common Lisp** dialect), **SCOOPS**. They have been brought to almost the same "purity of ideas" of object-oriented programming as the **Smalltalk** language, but the ability to use other means of the **LISP** language in parallel allows writing more effective programs. In the USA, a standard of object-oriented programming for the **Common Lisp** dialect is being developed **- CLOS (Common Lisp Object System)**.

Take a look at the combined family tree of **LISP** and **Smalltalk**:

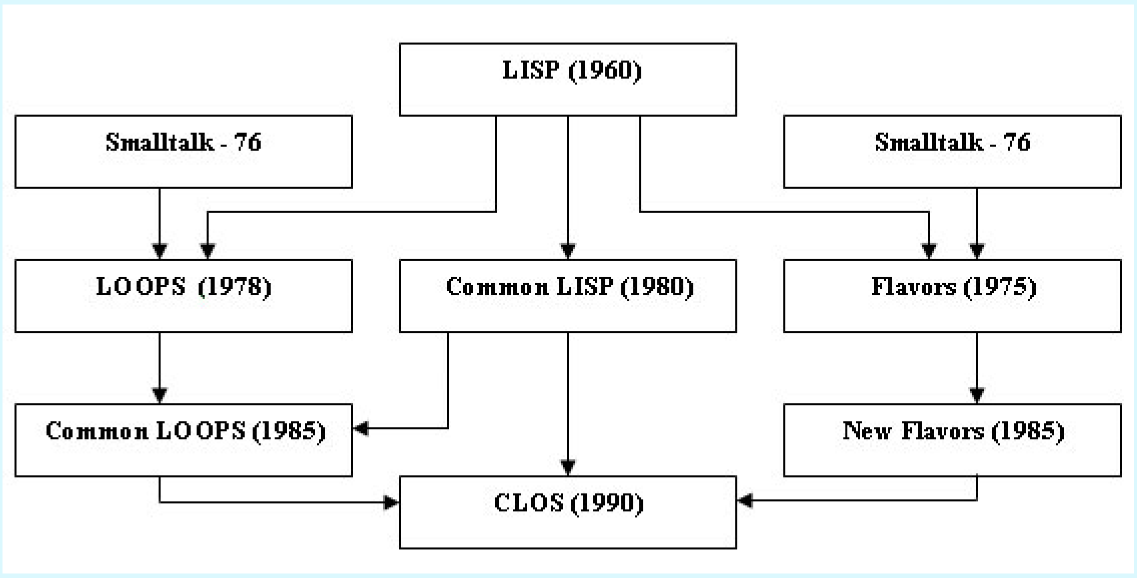


Fig. 1. Genealogical tree

Let's decipher some of ***the acronyms*** shown in the diagram:

- **LOOPS (Lisp Object Oriented Programming System)** is the predecessor of the **CLOS** language on **the Xerox LISP** machine.
- **CLOS** (**Common Lisp Object System**, pronounced "See-Loss" or "Closs") is an object-oriented programming standard in the **Common Lisp** language. It is based on **Symbolics FLAVORS** and **Xerox LOOPS**.

- **PCL (Portable Common Loops)** is a compact implementation of the **CLOS** language.

Let's talk a little about ***the history of the development*** of the functional programming paradigm.

1931: American mathematician and logician ***Alonzo Church*** describes the lambda calculus, which operates on three elements: symbols, brackets, and function notations expressed by the Greek letter "lambda". The lambda calculus forms the basis of the notation system of the **LISP** language.

1956: American mathematician ***John McCarthy*** **developed** the basics of the **LISP** language while working at the Summer Research Project on Artificial Intelligence (Dartmouth). Early implementations were made for **the IBM 704, IBM 7090, DEC PDP-1, DEC PDP-6**, and **DEC PDP-10**. **The PDP-6** and **PDP-10** had 18-bit addresses and 36-bit words, allowing **the CONS** cell to be stored in a single word, with simple instructions to extract **the CAR** and **CDR** portions of the cell. Early **PDP** machines had a small address space, which limited the size of **LISP** programs.

1960: John McCarthy's paper "Recursive Functions and Symbolic Computation" is published, providing the mathematical foundation for the **LISP** programming language, the main programming language in artificial intelligence research.

1960 – 1965: The emergence and distribution **of Lisp1.5 -** the first dialect of the **LISP** language.

1967: A native South African professor of mathematics and pedagogy at the Massachusetts Institute of Technology ***Seymour Peipert*** **created the LISP** -based language **LOGO** ("logo" in Greek for "word").

1969: ***Anthony*** **Hearn** and ***Martin*** **Griss** used **Standard Lisp** to make the **REDUCE** symbolic algebra system more compact for different architectures.

Late 60s: **Lisp1.5** spawns two main dialects: **Interlisp** and **MacLisp**.

Early 1970s: Development of specialized computers known as **Lisp** machines, designed specifically to speed up the execution **of LISP** programs. For example, **the Xerox D-series LISP** machinewas used for programs written in the **Interlisp-D** language.

1977: John Backus's Turing Lecture, devoted to a total criticism of the situation that had developed in programming. Here is a long excerpt from the work [1, pp. 54-56] (see also [2]).

Oddly enough, Turing's lecture was devoted to a total criticism of the situation that had developed in programming - and precisely in those sections for which he had been awarded the prestigious prize for his successful work:

"Imperative programming languages are becoming more and more cumbersome, but not more powerful. They are "fat" and "flabby" because of their fundamental inherited flaws: the primitive, word-by-step programming style inherited from their common ancestor, the von Neumann computer; the vicious coupling of state transition semantics; the separation of programming into the world of expressions and the world of operators; the inability to effectively use the power of combining forms to build new programs from existing ones; and finally the paucity of useful mathematical properties for reasoning about programs."

"While language bloat has increased their capabilities only marginally, small and elegant languages like Pascal remain popular. But we desperately need a powerful methodology to help us think about programs, and no imperative language comes close to fulfilling that goal."

Of course, it is surprising to hear such a speech from the lips of a person who stands at the origins of the von Neumann style of programming, the head of the group that developed the first imperative high-level programming language - Fortran, who came up with fundamental constructions that have been inherited from one language to another and have captured the entire programming world today. It is not for nothing that the citation index of this article has crossed all boundaries. For more than twenty years now, every self-respecting specialist has considered it his duty to refer to this work when touching on the problems of programming foundations. It is a great pity that this article, for a number of reasons, has not yet been published in Russian.

What made the master of the von Neumann style "betray" his brainchild? Alas, there were enough reasons for this. Already in the late 60s, a number of leading specialists began to sound the alarm about the crisis phenomena in programming (this is mentioned in the book by D. Gris "The Science of Programming"). The real scourge was software errors. It is not so bad when errors occur in game programs. It is much worse if an error is discovered in the software support of a spacecraft during its flight (unfortunately, such precedents have occurred - and with a tragic ending). Another manifestation of the crisis was the incomparable growth rates of productivity and other characteristics of computers on the one hand and the growth of programmer productivity on the other. Over 30 years of programming development, programmer productivity has increased by 10-15 times at most.

In his criticism, Backus constantly emphasizes that imperative languages have adopted the main vices of their "spiritual ancestor" - the von Neumann computer.

"In its simplest form, a von Neumann computer consists of three parts: a central processing unit (CPU), a memory, and a tube connecting them that can transfer one word at a time between the CPU and the memory... I prefer to call this tube the von Neumann **bottleneck...**

The assignment operator is the "bottleneck" of programming languages, which forces us to think in terms of "word-by-step" in exactly the same way as the bottleneck of a computing machine does."

And as ***a conclusion***:

"The problem is not only and not so much the bottleneck for the transmission of information, but also (and this is much more important) the bottleneck of the intellectual, in which we have become firmly entrenched with our "word-by-step" reasoning instead of rising to thinking in more general conceptual categories of the task being solved at the moment."

In his opinion, there is only one way out: ***using non-standard programming***. In his Turing lecture, Backus first systematically presents the idea of the **FP** style **(Functional Programming Style)**.

Late 70s.

The Macsyma group at MIT designed the **NIL (New Implementation of Lisp) language, a Lisp** language for the **VAX** family of computers.

National Laboratory (**Stanford and Lawrence Livermore**) designed **S-1 Lisp** for the **Mark IIA** supercomputer.

Franz Lisp (a dialect of the **MacLisp** language) began running on **Unix** machines.

G.Sussman and G.Steele **developed** the Scheme language **- a** simple dialect of **Lisp**.

"Advent" of object-oriented programming in **LISP**. The **Flavors** system was developed for **the Lisp** machine at MIT. **LOOPS (Lisp Object Oriented Programming System)** was implemented on the **Xerox** computer.

1981, April. The appearance of the **Common Lisp** language, containing common aspects of various versions of the **LISP language (Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, Scheme)**.

1984. Publication of the book **"Common Lisp: The Language"** by ***Guy* L. Steele**.Common **Lisp** becomes the **de facto** standard.

1990 G.L. Steele publishes the book **"Common Lisp: The Language and Edition"**, which includes information about **CLOS**.

1992. X3J13 developed a draft of a proposed American National Standard **for** Common **Lisp**. This document is the first official "successor" to G.L. Steele's book **"Common Lisp: The Language"**. X3J13 is a subcommittee of the ANSI (American National Standards Institute) committee that worked on **the ANSI** standardization of **the Common Lisp** language.

---

(1) Mathematical logic in programming: Collection of articles from 1980-1988: Translated from English. - Moscow: Mir, 1991. - 408 p.
(2) Lectures of Turing Award laureates for the first twenty years. 1966-1985. - Moscow: Mir, 1993. - 560 p.

In the next step we will look at ***the production paradigm***.

Step 52.
Production paradigm

In this step we will look at ***the production paradigm***.

First, as always, information from explanatory dictionaries [1-4].

Rule-Oriented Programming *Languages* (ROPLs) are a class of declarative programming languages based on an approach to programming in which a program is specified by a set of rules (productions) without explicitly specifying the sequence of their application. Rules contain either a condition and actions that must be performed if this condition is true, or a condition and a set of other conditions sufficient for this condition to be true.

D. Gris [5, p. 131] writes: "A production language is a programming language intended for writing syntactic recognizers. The program basically consists of a sequence ***of productions***. The use of this word here should be considered rather unfortunate, since this word is usually associated with another meaning, denoting the rules of substitution in grammar. These new productions would be better called ***reductions***, since they are (usually, but not always) used to reduce sentences to the initial symbol of the grammar. We will not, however, invent a new term and will stick to the word "production", which is constantly used in the literature. A production language is usually part of a system for constructing translators (i.e., a system intended for the implementation of such programs as assemblers and compilers."

V.N. Agafonov in the collection [6, p.331] uses the terms ***"term substitution system", "reduction system"***.

Let's talk about ***the history of the development*** of the production (reduction) programming paradigm.

1948. Construction by A.A. Markov of the theory of normal algorithms, later presented in the book "Theory of Algorithms", 1954.

1961. R. Floyd first described the production language in the article "A descriptive language for symbol manipulation", JACM, 8 (Oct. 1961), 579-584.

1964. The appearance of the extended version of the **SNOBOL** language.

1968. Creation of ***the Refal language by V.F. Turchin*** [7,8].

---

(1) Borkovsky A.B. English-Russian Dictionary of Programming and Computer Science (with Explanations). - Moscow: Russian Language, 1989. - 335 p.
(2) Zamorin A.P., Markov A.S. Explanatory Dictionary of Computer Science and Programming: Basic Terms. - Moscow: Russian Language, 1988. - 221 p.
(3) Explanatory Dictionary of Computing Systems / Ed. by V. Illingworth, E.L. Glazer, I.K. Pyle. - Moscow: Mashinostroenie, 1989. - 568 p.
(4) Pershikov V.I., Savinkov V.M. Explanatory Dictionary of Computer Science. - Moscow: Finance and Statistics, 1991. - 543 p.
(5) Gris D. Design of Compilers for Digital Computers. - M.: Mir, 1975. Pp. 367-376, ch.16.
(6) Logical programming. - M.: Mir, 1988. -368 p.
(7) Basic REFAL and its implementation on computers. / TsNIPIASS. - M., 1977. - 238 p.
(8) Turchin V.F. Basic REFAL. Description of the language and basic programming techniques. - M.: TsNIPIASS, 1974. - 258 p.

In the next step, we will focus on ***programming paradigms used in artificial intelligence tasks***.

Step 53.
Programming paradigms in artificial intelligence tasks

In this step, we will look at ***programming paradigms used in artificial intelligence problems***.

The first artificial intelligence programming languages were developed back in the early 1960s, when research in the field of artificial intelligence began and it was noted that the tasks in this area differ significantly from both computational tasks and business information processing tasks. They are associated with the processing of symbols (in the broad sense of the word), which is difficult to represent as a single, strictly defined algorithm, so two principles are ***the basis of artificial intelligence programming languages:***

- ***orientation towards symbolic processing*** and
- ***declarative approach to programming*** [1].

Among the programming languages of artificial intelligence, the leading place is occupied by **LISP** and **PROLOG**. In the USA and Japan, despite the widely known project of creating fifth-generation computers, where the use of the **PROLOG language was declared, LISP** dominates, in Europe, preference is given to the **PROLOG** language. These languages are classic representatives of the declarative (**PROLOG**) and functional (**LISP**) approaches to programming. In the process of research in the field of artificial intelligence, new approaches and languages have emerged. However, by now only a few of these languages have received at least limited distribution - this includes **POP, Tablog, LOGO, LogLisp, SNOBOL, *Refal***. Only **PROLOG, LISP** and, to some extent, **OPS5** have been able to establish themselves on the market.

LISP (or rather, its modern dialects) is not the only language used for artificial intelligence tasks.

For example, **POP-2** is a programming language developed at the University of Edinburgh (UK) with a focus on artificial intelligence research. **POP-2** provides capabilities for manipulating related data structures - just like the **LISP** language, but has a simpler procedure structure, so it has become more accessible to programmers who previously worked with the **ALGOL** language.

Already in the mid-1960s, i.e. at the stage of the formation of the **LISP** language, languages were being developed that offered other conceptual foundations. The most important of them in the field of symbolic information processing are **SNOBOL**, developed at Bell Labs, and ***Refal***, created at the IPM of the USSR Academy of Sciences.

The SNOBOL language is a string processing language in which the concept of pattern searching first appeared and was implemented to a fairly full extent. The **SNOBOL** language was one of the first practical implementations of a developed production system. The most famous and interesting version of this language is **SNOBOL-IV**. Here, the technique of specifying patterns and working with them significantly outpaced the needs of practice. Perhaps this, as well as the policy of active implementation of the **LISP** language, prevented the widespread use of the **SNOBOL** language in the field of artificial intelligence. In essence, it remained a "branded" programming language, although the concepts of the **SNOBOL** language certainly influenced both **LISP** and other programming languages for artificial intelligence.

The Refal language is based on the concept of a recursive function defined on a set of arbitrary symbolic expressions. When processing symbols, the concept of pattern search, characteristic of the **SNOBOL** language, is actively used. Thus, ***Refal*** has absorbed the best features of the most interesting symbolic information processing languages of the 60s. At present, the ***Refal-5*** language is used to automate the construction of translators, automatic transformation systems, and, like the **LISP** language, as an instrumental environment for the implementation of knowledge representation languages.

OPS5 is a language that has become widespread mainly on minicomputers and large machines. Unlike all other artificial intelligence programming languages, **OPS5** is not a universal, but a specialized programming language that allows achieving the maximum ratio of ***"efficiency/simplicity"*** when solving problems of developing expert systems. It is known that systems implemented with this language must fit into fairly strict syntactic frameworks, but the experience of creating many complex expert systems shows that the means of the **OPS5** language are sufficient in most cases.

A certain amount of experience in implementing intelligent systems has already been accumulated, and it can be said that certain programming paradigms have been formed in the field of artificial intelligence. The main ones, apparently, are ***the functional, logical and production*** paradigms.

Of course, in modern practice of intelligent programming, individual paradigms are rarely used in their pure form. And as examples of the fusion of different paradigms, one can cite not only individual languages, but also a whole approach to programming intelligent systems based on the production paradigm, within which the interaction of rules-productions can rely on the functional paradigm, and the unification of the conditions of applicability of rules - on the logical one.

Thus, the modern stage of programming intelligent systems is characterized by a tendency to mixed use of different paradigms.

An analysis of existing symbolic information processing languages, their use for the implementation of intelligent systems, and a comparison of the development trends of these languages allow us to make several observations.

1. It can be assumed that **LISP** will remain the main language for implementing intelligent systems for a considerable time.
2. In the near future, we can expect the emergence of languages that have incorporated the best features of the **LISP** language and, for example, **PROLOG**.
3. There is a clear trend towards creating parallel versions of symbolic information processing languages and languages for programming artificial intelligence tasks.
4. **It appears that the object-oriented paradigm and, as a consequence, languages like Smalltalk** will be the fastest moving knowledge-based systems. At the same time, it is possible to assume that **LISP** (as it has already done in the past) will try to make another mutation and thereby regain its leading position.
5. Languages such as **LISP, PROLOG, *Refal*** (as well as all sorts of modifications and "mixtures" of these and/or other symbolic processing languages) will increasingly give up their positions at the level of knowledge engineers to ***special knowledge representation languages***, while remaining the tools of system programmers.

(1) Artificial Intelligence: - In 3 books. Book 3. Software and hardware: Handbook / Edited by V.N. Zakharov, V.F. Khoroshevsky. - M.: Radio and Communications, 1990. - 368 p.

We have finished looking at the basic programming paradigms. From the next step we will begin to get acquainted with the features of the **LISP** language in more detail, in particular, we will focus on ***the assignment function***.

Step 54.
Assignment functions

In this step, we will look at ***functions that are equivalent to assignment operators in imperative programming languages***.

The assignment operator does not play such a significant role in LISP programming as it does in imperative programming languages. Many **LISP** programs are written without a single assignment operator; the same effect is achieved indirectly, using recursion and parameter passing.

A distinction is made between global and local values of atoms.

Local values are bound to an atom only for the duration of some functions. ***Global values*** of atoms can be set using the so-called ***assignment functions*** **SET** and **SETQ**. Then, if an atom with a global value acts as a formal parameter of some function, it is bound to a new value during lambda transformation. After that, the atom returns to its previous global value.

The SET function binds the value of the **X** argument to the calculated value of the **S** expression. The syntax of the function is:

```
        (SET X Y)
```

where **X** is an atom, **Y** is an **S** expression.

For example:

```
    $ (SET 'FOO '(A B C))     $ (SET 'PET 'DOG)
    (A B C)                   DOG
    $ FOO                     $ (SET PET 'ROVER)
    (A B C)                   ROVER
                              $ DOG
                              ROVER
                              $ PET
                              DOG
```

Note that **(SET X Y)** is equivalent to **X := Y** in imperative programming languages.

Note that the **SET function *evaluates both arguments***, i.e. it can be used to associate a **Y** value with a name that is also obtained by evaluation. For example:

```
    $ (SET (CAR '(A B C)) 25)
    25
    $ A
    25
```

You can also associate a symbol with its value using the **SETQ** function. This function differs from the **SET** function in that it only evaluates its second argument. The letter **Q (Quote)** in the function name reminds you of the automatic blocking of the first argument.

Thus, **(SETQ A B)** is equivalent to **(SET (QUOTE A) B)**.

*Note: The **InterLISP** implementation has a function **SETQQ** that blocks the evaluation **of both** arguments.*

Example 1. Generation of random integers not greater than 8.

```
(DEFUN RANDOM (LAMBDA (SEED)
      (SETQ SEED (MOD
                     (+ 2113233
                           (* SEED 271821)) 9999991))
      (MOD SEED 8)
))
```

The SET and **SETQ** functions also have ***a side effect***. The effect of the function is to form a bond between an atom and its value. The atom remains bound to a certain value until that value is changed.

Let us recall that functions with a side effect are called ***pseudo-functions*** in **LISP**. We will still use the concept of a function for both functions and pseudo-functions, unless there is a special need to emphasize the presence of a side effect.

We will graphically represent the result of the function **(SETQ LIST '(A B C))** as follows:

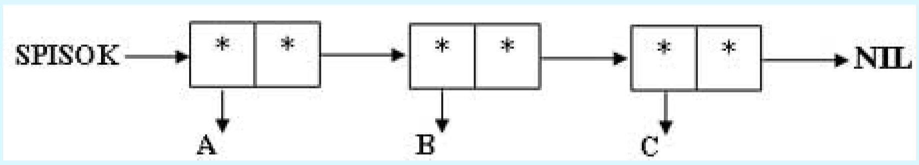


Fig. 1. Illustration of list definition

Example 2. Let's assume that we have built two lists:

```
    $ (SETQ RELEASE '(B C))
    (B C)
    $ (SETQ HVOST '(A B C))
    (A B C)
```

Let's create a new list:

```
    $ (SETQ RES (CONS GOLOWA HVOST))
    ((B C) A B C)
    $ RES
    ((B C) A B C)
```

Let's depict the result of the construction in the figure:

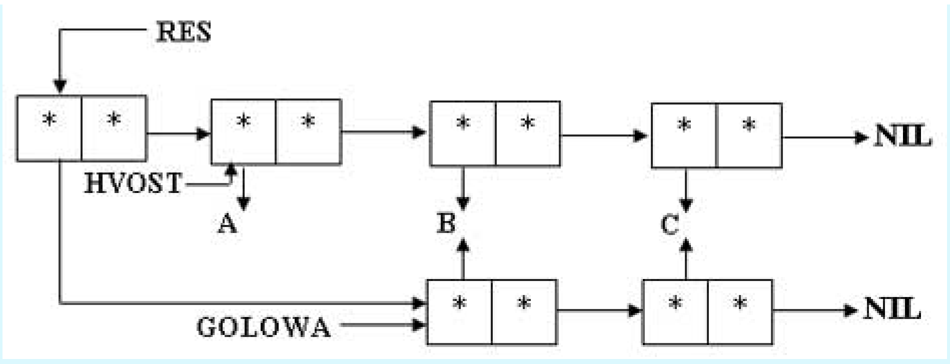


Fig.2. Construction result

It is easy to see that one cell can be pointed to by one or more arrows from the list cells, but ***only one arrow can come out of each cell field***. If there are several pointers to a cell, then that cell will define the common subexpression.

Depending on the construction method, ***the logical and physical structure*** of the two lists may be different.

Example 3.

```
$ (SETQ LIST1 '((B C) A B C))
((B C) A B C)
```

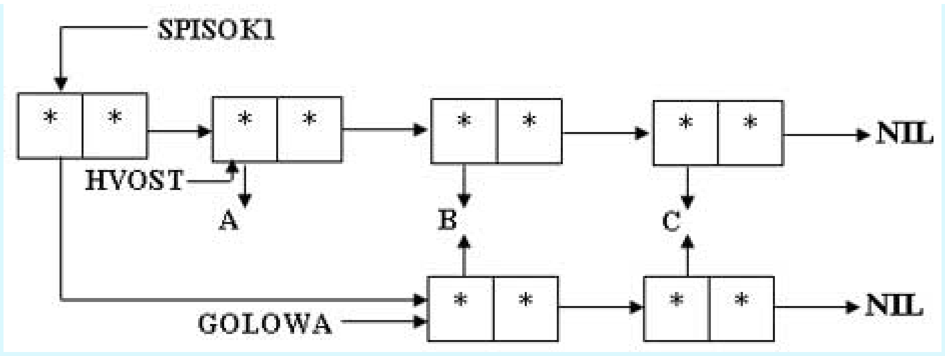


Fig.3. Definition of the list **LIST1**

```
$ (CAR LIST1)
(B C)
$ (CDDR LIST1)
(B C)
```

Logically identical lists were obtained.

```
$ (SETQ BC '(B C))
(B C)
```

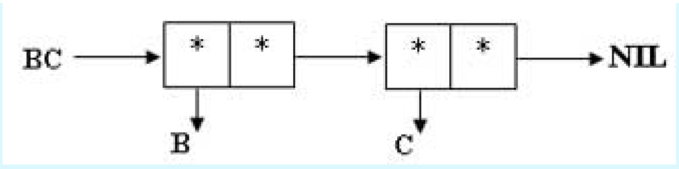


Fig.4. Definition of the **BC list**

```
$ (SETQ A B C (CONS A B C))
```

```
    (A B C)
```

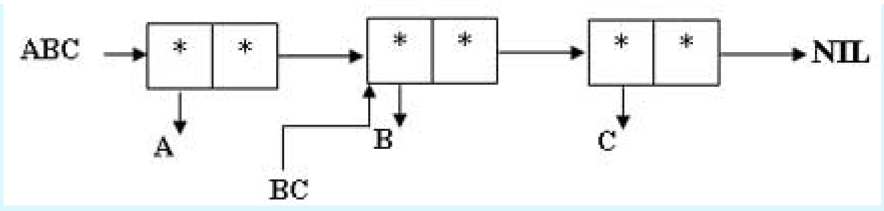


Fig.5. Definition of the **ABC list**

```
    $ (SETQ LIST2 (CONS B C A B C))
    ((B C) A B C)
```

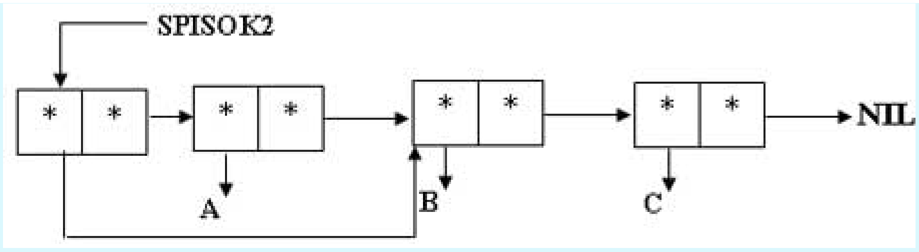


Fig.6. Definition of the **LIST2 list**

Thus, in examples 3 and 4 we got ***two logically identical lists with different physical structures***.

The logical structure is always topologically in the form of a binary tree, while ***the physical structure*** can be an acyclic graph (branches can converge again, but can never form closed cycles, i.e. point backwards).

---

*Note: The **muLISP-85** version has another interesting assignment function: **PSETQ**.*

*The operation of the function **(PSETQ SYMBOL1 FORM1... SYMBOLN FORMN)** is identical to that of the function **SETQ** except that all **FORMj** arguments are evaluated before any assignments are made to the **SYMBOLj** atoms. The name **"PSETQ"** is an abbreviation for **"Parallel SETQ"**.*

For example:

```
    $ (SETQ SUM 5)
    5
    $ (PSETQ SUM (+3 4) SQR (* SUM SUM))
    25
    $ SUM
    7
    $ SQR
    25
```

*Moreover, in this version the **SETQ** function is more powerful than in the later versions.*

The function **(SETQ SYMBOL FORM)** evaluates **FORM, takes SYMBOL** as the result, and returns the result. Note that **SETQ** is a special form and that **SYMBOL** is not evaluated.

If **SYMBOL** is not a symbolic atom, **SETQ generates a *"Non-symbolic argument"*** error trap.

If the **SETQ** function is specified with more than two arguments **((SETQ SYMBOL1 FORM1 SYMBOL2 FORM2 ... SYMBOLN FORMN))**, then the forms are evaluated and the values are assigned sequentially. If an odd number of arguments are specified, the last symbol is taken as **NIL. The SETQ** function returns the new value ***of the last*** symbol of **SYMBOLN**. For example:

```
    $ (SETQ FOO '(D E F))       $ (SETQ SUM (+3 4) SQR (* SUM SUM))
    (D E F)                     49
    $ FOO                       $ SUM
    (D E F)                     7
    $ (SETQ FOO (CDR FOO))      $ SQR
    (E F)                       49
    $ FOO
    (E F)
    $ (SETQ SUM 5)
    5
```

Here is a macro for this function:

```
    (DEFMACRO SETQ (SYM OBJ)
       (LIST 'SET (LIST 'QUOTE SYM) OBJ) )
```

In the next step we will look at issues related to ***memory management***.

Step 55.
Memory Management

In this step we will look at ***the functions and variables used for memory management***.

Calculations can result in ***garbage*** in memory - these are structures that can no longer be referenced. This happens when the calculated structure is not saved using functions like **SETQ**, or when a reference to an old value is lost as a side effect of a new call to **SETQ** or another function. For example:

```
    $ (SETQ LIST ((This will become garbage) And this part will not))
```

Here is a graphical representation of the constructed list:

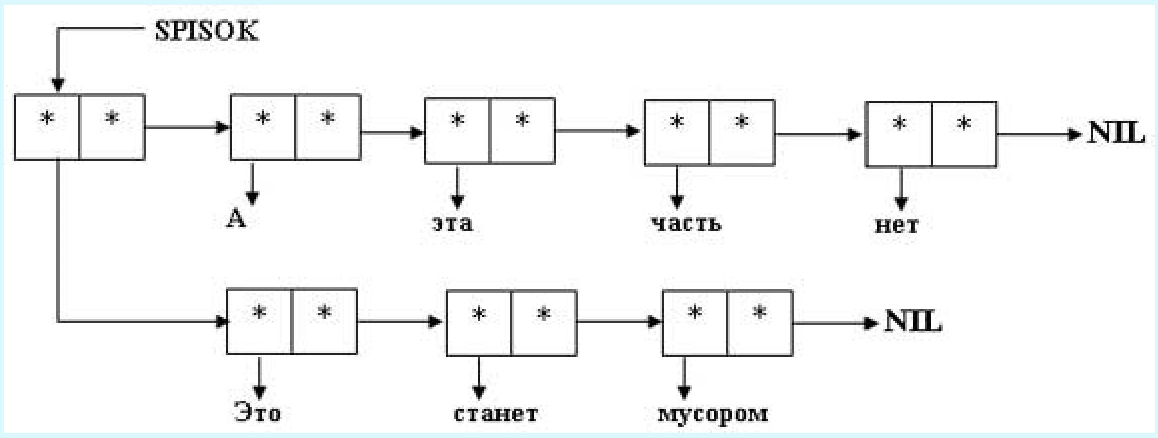


Fig. 1. Graphical representation of the list

After assigning a new value:

```
    $ (SETQ LIST (CDR LIST))
```

it will no longer be possible to “get” to three list cells. They say that these cells ***have become garbage***.

This is quite obvious if you look at the graphical representation of the **LIST** list.

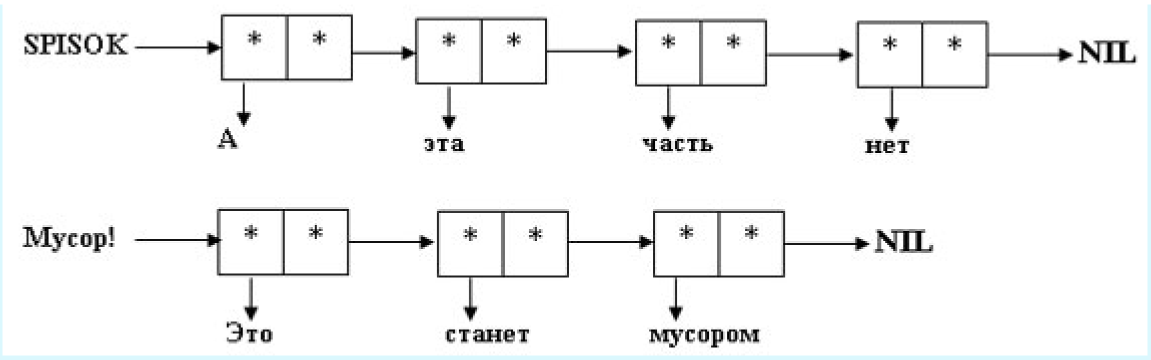


Fig.2. Graphical representation of the list

Garbage also occurs when ***the result of a calculation is not associated with any variable*. For example, the value of a CONS** function call:

```
   $ (CONS A (LIST B))
```

is only displayed on the display screen, after which the corresponding structure will remain in memory as garbage.

To reuse memory that has become garbage, **LISP** systems have a special ***garbage collector*** that is ***automatically*** launched when there is little free space left in memory. The garbage collector goes through all the cells and collects the cells that are garbage into a list of free memory so that they can be used again.

With limited user-accessible memory, garbage collection often significantly slows down interactive access to the computer. This prompts Loos to note: "Perhaps the only way to optimize the garbage collection process and limit its impact is to develop a special algorithmic language that ties together all aspects of computation, including garbage collection."

The RECLAIM function controls the garbage collector and returns the amount of free memory in the data area. Since memory management in **muLISP** is completely automatic, **RECLAIM** is used only to determine the amount of free memory or to minimize the number of "garbage collections" during program execution. Memory reallocation is also performed if necessary. The total number of bytes available for atom, vector, pointer, and stack areas is returned. The function syntax is:

```
   $ (RECLAIM)
```

*Note: Dynamic automatic memory management gives **muLISP-85** a big advantage. It frees the programmer from having to pre-allocate the available memory for a task. Such pre-allocation is very difficult (if possible at all!) for most AI tasks.*

During the initialization phase of the **muLISP-85** system, the amount of I/O memory available to the system is calculated. The memory is then divided into ***four non-overlapping data areas***.

The atomic region is a memory area for storing information about atoms and numbers (the four pointer elements required for each atom and number).

The vector area is a region of memory for storing **the P-name**s of atoms and the numeric binary vectors of each number.

The pointer area is the memory area allocated to the two pointer elements required for each dotted pair and **the D *code*** required for each function definition. Since the dotted pair is the basic, simplest data structure **in muLISP**, the pointer area is usually the largest of all the data areas.

The stack area is the memory area for the control stack and the variable stack. These two stacks are located at opposite ends of the stack area.

When **muLISP-85** runs on an **Intel 8086** family computer, the memory is divided into 64K segments. This places a limit on the size of each **muLISP** data area.

Therefore, the memory available for the vector area is 128K: 64K for P-names and 64K for vectors of numbers.

The memory area available for pointer elements of data objects (i.e. atoms, numbers, and dotted pairs) is 25K.

The memory area available for **D** codes and stacks is 64K.

The memory area available for initial expressions and user-defined machine codes is 64K.

Therefore, the **muLISP** system requires 512K to run, plus the memory required by the operating system.

The muLISP-85 system uses ***a two-pass*** (mark-sweep) garbage collection algorithm.

The first scan of the algorithm consists of marking all active data objects. These are those objects that are accessed by chaining with pointer elements, starting from the elements of the value list and properties of all symbols in the system, or from the variable stack, or from **D** codes. Auto-referenced symbols that have no properties and no current function definitions are not marked. Such symbols are automatically removed from the list during the second scan.

During ***the second pass*** of garbage collection, all marked data objects are compacted and collected at one end of the corresponding data area. This allows the remnants of the available data areas to be saved for newly created objects.

After garbage collection, any one (or more) of **muLISP-85's** four data areas may have insufficient free memory for programs to continue executing, even though other data areas have sufficient free memory. If this situation occurs, ***the data areas are re-allocated*** by dividing the areas, adding the missing memory to one or more areas. (However, the size limits for each data area, as described above, must be respected during allocation.)

Although garbage collection and data relocation occur automatically, their occurrence is not unnoticeable to the user, as they cause a short pause in the operation of programs.

The exact amount of time for garbage collection and reallocation depends on the amount of data in the system. Garbage collection typically takes less than a second. Likewise, reallocation of data areas typically takes less than a second. This should not really be a concern for the user, but it is a consideration when developing real-time systems using the **muLISP-85 system.**

A phenomenon known as **"Thrashing"** ***occurs*** when the system is forced to spend an unexpected amount of time collecting garbage for a "very small" return of data space. **Thrashing** is indicated by a significant increase in the execution time of a task. This problem can be resolved by increasing the size of the computer's memory (up to 512K) and/or modifying the program to reduce its memory requirements.

Below we describe some of the functions and system variables used in memory management.

1. If **N** is a positive integer, the **ALLOCATE N** function frees **N** bytes of memory in the 8086 code segment following the **muLISP** machine code and returns the base address of the newly allocated memory. If **N** bytes cannot be freed, the **ALLOCATE** function returns **NIL**.

The (ALLOCATE) function returns the current base address of the start of the free region in the 8086 code segment following the **muLISP** machine code. For example:

```
   $ (ALLOCATE 100)    $ (ALLOCATE)
   32400               32512
```

2. If the **RECLAIM** system variable is **NIL**, informational statistics are displayed on the console before and after all garbage collectors and memory reallocations have been performed.

The first line of statistics shows what area is available for work before the garbage collector runs.

The second line shows the available area after the collector has run.

Lines starting with "**GC**" indicate garbage collector statistics. Lines starting with "**RA**" indicate data area reallocation process statistics.

The format of the statistics line is as follows:

```
GC : nnnn aaaa/aaaa vvvv/vvvv pppp/pppp ssss/ssss tttt/tttt
```

nnnn, aaaa, vvvv, pppp, ssss, and **tttt** are hexadecimal numbers. **nnnn** is the number of garbage collections since **muLISP** started executing.

The first numbers **aaaa, vvvv, pppp** and **ssss** in each pair of numbers are the sum of free memory in bytes in the atom, vector, pointer and stack areas respectively. The first number **tttt** of the pair is the total sum of free areas (the sum of **aaaa, vvvv, pppp** and **ssss**).

The numbers following the "/" are the total volume of the corresponding areas, including both free and occupied areas.

Each pointer location in the pointer, stack, and atom areas requires two bytes. One byte is required for each character in **the P-name**, two bytes to accommodate the length of **the P-name**. Numeric vectors use as many words as are needed to accommodate the number in binary.

This statistic is highly dependent on how **muLISP** works, and is not user-dependent. This statistic also appears when a **"Memory full"** error occurs, and indicates how much memory is full. For example:

```
   $ (SETQ RECLAIM NIL)
   NIL
   $ (LOOP  (OBLIST))
GC: 0001 2258/2600 1C8C/1C8E 00000/119C0 1AAA/1AC0 0598C/1770E
GC: 0001 2258/2600 1C8C/1C8E 117D0/119C0 1AAA/1AC0 1715C/1770E
RA: 0001 09C0/0D68 09C2/09C4 13CFE/13EEE 20DE/20F4 1715C/1770E

GC: 0002 09C0/0D68 09C2/09C4 00002/13EEE 20DE/20F4 03460/1770E
GC: 0002 09C0/0D68 09C2/09C4 13E1A/13EEE 20DE/20F4 17278/1770E

GC: 0003 09C0/0D68 09C2/09C4 00002/13EEE 20DE/20F4 03460/1770E
GC: 0003 09C0/0D68 09C2/09C4 13E1A/13EEE 20DE/20F4 17278/1770E
```

The three lines of GC 1 statistics mean that a reallocation has occurred. More memory has been allocated for the pointer area because **OBLIST** "requires" new dot pairs. Subsequent GCs do not perform reallocations because the sizes of the data areas have already been optimized. However, if the program has changed and "accessed" the atom area, another reallocation may occur to increase the size of that area at the expense of others.

The numbers given above depend on the memory size of the computer; here they are given for the 8086/8088 version of **muLISP-85**.

3. Constructor functions usually "require" new dotted pairs. If the value of the system variable ***FREE-LIST*** is an atom, then the area for new dotted pairs is taken from the memory area called the pointer area. If the entire pointer area has been exhausted by creating previous dotted pairs, then the garbage collector is automatically started to restore the areas for dotted pairs.

   However, if the ***FREE-LIST*** value is a dotted pair, then that dotted pair is used when a new dotted pair is required by the **muLISP-85 system.**

   The example adds 6 (not 9) dotted pairs to ***FREE-LIST***:

```
    $ (SETQ JUNK '(A B (C D E) 3 4 5))
    $ (SETQ *FREE-LIST* *NCONC JUNK *FREE-LIST*)
```

4. If **ADDRESS** is zero or a positive integer less than the memory area of the microprocessor on which **muLISP-85** is running, the **(MEMORY ADDRESS)** function transfers a byte (8-bit value) to **ADDRESS**.

   If **VALUE** is zero or a positive integer less than 256, the **(MEMORY ADDRESS VALUE)** function changes the byte at **ADDRESS** to **VALUE** and returns the original value of the byte.

   If **FLAG** is not **NIL**, then **(MEMORY ADDRESS VALUE FLAG)** transfers ***the word*** (16-bit value) to **ADDRESS**. If **VALUE** is zero or a positive integer less than 65536, then **(MEMORY ADDRESS VALUE FLAG)** stores the **VALUE of** the word at **ADDRESS** and returns the original value of the word.

   For example:

```
    $ (SETQ *PRINT-BASE* 16 *READ-BASE* 16)
      10                          ; Hexadecimal I/O
    $ (MEMORY 12)
      0                           ; Byte at 12H
    $ (MEMORY 12 NIL T)
      0F000                       ; Word at 12H
    $ (MEMORY 12 5)
      0                           ; Store byte 5 at 12H and return previous value
    $ (MEMORY 12)
      5                           ; Byte at 12H
    $ (MEMORY 12 NIL T)
      0F005                       ; Word at 12H
    $ (MEMORY 12 0)
      5                           ; Store byte 0 at 12H and return previous byte
    $ (SETQ *PRINT-BASE* 0A *READ-CHAR* 0A)
                                 ; Decimal I/O
```

   Please note that the **MEMORY** function should be used with extreme caution, as there is no protection against damaging the **muLISP-85** interpreter or the operating system.

5. **The CSMEMORY** and **DSMEMORY** functions are available only when **muLISP-85** is running on an **Intel 8086** microprocessor family computer. These functions are similar to **MEMORY**, except that the first argument is a sixteen-bit segment field rather than a twenty-bit physical address. The first field of **CSMEMORY** is the code segment. In **DSMEMORY,** the first field corresponds to the data segment.

   If **OFFSET** is zero or a positive integer less than 65536, the **(CSMEMORY OFFSET)** and **(DSMEMORY OFFSET)** functions move the byte to the field at offset **OFFSET**.

If **VALUE** is zero or a positive integer less than 256, the **(CSMEMORY OFFSET VALUE)** and **(DSMEMORY OFFSET VALUE)** functions write the value of the **VALUE** byte to the field at offset **OFFSET** and return the original value of the byte.

If **FLAG** is not **NIL**, the **(CSMEMORY OFFSET NIL FLAG)** and **(DSMEMORY OFFSET NIL FLAG)** functions return the word in the field with offset **OFFSET**.

If **VALUE** is zero or a positive integer less than 65536, and **FLAG** is not **NIL**, the **(CSMEMORY OFFSET VALUE FLAG)** and **(DSMEMORY OFFSET VALUE FLAG)** functions change the word's **VALUE** in the field at offset **OFFSET** and return the word's original value. For example:

```
    $ (SETQ *PRINT-BASE* 16 *READ-BASE* 16)
                               ; switch to hexadecimal I/O
    $ (CSMEMORY 100)
    0E9                        ; returns Byte in CS:100H
    $ (CSMEMORY 100 NIL T)
    3E9                        ; returns Word in CS:100H
    $ (CSMEMORY 100 41)
    0E9                        ; stores 41H to CS:100H
    $ (CSMEMORY 100)
    41                         ; returns byte in CS:100H
    $ (CSMEMORY 100 NIL T)
    341                        ; returns word in CS:100H
    $ (CSMEMORY 100 0E9)
    41                         ; restores original byte value
    $ (SETQ *PRINT-BASE* 0A *READ-CHAR* 0A)
                              ; back to decimal I/O
```

We warn you that the **CSMEMORY** and **DSMEMORY** functions should be used with caution, since there is no protection against destruction of **muLISP** programs !

6. If **PORT** is zero or a positive integer less than 65536, the **(PORTIO PORT)** function inputs and returns the byte value from **PORT**.

If VALUE **is** zero or a positive integer less than 256, the **(PORTIO PORT VALUE)** function passes the byte **VALUE** to **PORT** and returns **NIL**.

If **FLAG** is not **NIL**, then the function **(PORTIO PORT NIL FLAG)** inputs and returns the value of the word from **PORT**.

If **FLAG** is not **NIL**, and **VALUE** is zero or a positive integer less than 65536, the **(PORTIO PORT VALUE FLAG)** function passes the word **VALUE** to **PORT** and returns **NIL**.

This function is used to send byte or word values to output ports and receive byte or word values from input ports. This can be used in **muLISP** programs when it is necessary to communicate with external devices such as sensors, robots, etc. Example:

```
    $ (SETQ *PRINT-BASE* 16 *READ-BASE* 16)
                               ; Hexadecimal I/O
    $ (PORT 72)
    <?>                        ; enter a byte from port 72H
    $ (PORT 72 10)
    NIL                        ; output 10H to port 72H
    $ (SETQ *PRINT-BASE* 0A *READ-CHAR* 0A)
                              ; Decimal I/O
```

7. The pointer elements included in objects are stored in separate data segments at the same field addresses. The function **(LOCATION OBJECT)** returns the first field (**Offset**) of the **OBJECT elements from**

the data segment database. Since all elements occupy exactly 2 bytes, the LOCATION function always returns an even integer.

Note that the **muLISP** system automatically performs garbage collection and memory reallocation, which can change the location of an object. The **LOCATION** function is primarily intended for compilers that generate machine codes and need to know how symbols are laid out in memory. For example:

```
    $ (LOCATION NIL)
    256
    $ (LOCATION 2)
    3458
    $ (LOCATION '(A. B))
    9376
```

8. The function **(REGISTER N M)** places **M** into the **muLISP pseudo-register N** if **M** is a non-negative integer less than 65536, and returns **NIL**.

   If **M** is omitted, the content of pseudo-register **N** is returned. By default, the value for all registers is taken to be 0. For example:

```
    $ (REGISTER 2)
    0
    $ (REGISTER 2 100)
    NIL
    $ (REGISTER 2)
    100
```

9. The function **(INTERRUPT N)** loads the values of the **muLISP** pseudo-registers into the microprocessor registers, executes an interrupt with number **N**, saves the register values issued during the interrupt in the pseudo-registers, and returns **NIL**.

   Note that the **INTERRUPT** function is available only when **muLISP** is running on an **Intel 8086** family of microprocessors.

   There are 10 pseudo-registers (0 through 9) in **muLISP. The correspondence between them and the 8086 registers is given in the following table:**

```
Pseudo register: 0  1  2  3  4  5  6  7  8  9
 Registers 8086: AX BX CX DX IF DI BP DS IS FLAG
```

The REGISTER and **INTERRUPT** functions provide communication between external machine language templates and **muLISP** programs via the 8086 microprocessor's 256 interrupt vectors. Computer technical recommendations and operating system manuals contain information about the service that interrupt vectors provide.

For example, interrupt number 33 (21H) is used by MS-DOS for system calls. If the **AH** register contains 44 (2AH), then (Interrupt 33) will return: year - **CX** (1980-2099), month - **DH** (1-12) and day - **DL** (1-31). For example:

```
    $ (SETQ *PRINT-BASE* 16 *READ-BASE* 16)
    $ (REGISTER 0 2A00)
    NIL
    $ (INTERRUPT 21)
    NIL
    $ (SETQ *PRINT-BASE* 0A *READ-BASE* 0A)
    $ (REGISTER 2)
    1985
    $ (DIVIDE (REGISTER 3) 256)
    (1. 5)
```

10. If **OFFSET** is a positive integer less than 65536, the function **(BINARY-LOAD DRIVE:NAME.TYPE OFFSET)** loads the file **NAME.TYPE** from the **DRIVE** device into the code segment at offset **OFFSET** and returns the number of bytes loaded.

    If **DRIVE** is missing, the current drive is used. If **OFFSET** is not in this area or the file is not found, **BINARY-LOAD** returns **NIL**.

    The BINARY-LOAD function is typically used in conjunction with the **ALLOCATE** function.

11. If **ADDRESS** is zero or a positive integer less than the address range of the microprocessor on which **muLISP** is running, and **N** is a positive integer less than 65536, then the function **(SNAPSHOT ADDRESS N)** returns an atom whose P-name consists of **N** bytes of memory starting at **ADDRESS**.

    If **SYMBOL** is a symbol, then the function **(SNAPSHOT ADDRESS SYMBOL)** places the bytes in **the P-name** of the symbol **SYMBOL** in memory, starting at address **ADDRESS**, and returns **T**.

    The SNAPSHOT function is especially useful for manipulating **the IBM PC's** video screen memory. For example:

```
    $ (SETQ *PRINT-BASE* 16 *READ-BASE* 16)
    $ (SNAPSHOT 0B0000 200)
    <First 512 bytes of PC video memory>
    $ (SETQ *PRINT-BASE* 0A *READ-BASE* 0A)
```

In the next step we will look at ***the simplest modifiers***.

Step 56.
Basic modifiers

In this step we will look at ***functions that modify structures***.

The LISP language has special functions that can be used to "tear apart" a structure and "glue" it together in a new way. Such functions are called ***structure-destroying*** or **modifiers**. Modifiers perform pointer redirection in **LISP data structures. Therefore, when using modifiers,** ***the result of their work*** is more important than the value they return.

The danger of using these operations arises from the possible existence of common sublists that the programmer may forget about. If one list is modified and is a sublist of another list, then that list will also be modified.

The effectiveness of these operations is due to two reasons:

1. no additional work is required to change the list;
2. When building lists, there is no need to make a lot of copying.

Here is a list of functions that are considered at this step:

Table 1. **The simplest modifiers**

Function	**Purpose**
(RPLACA OBJECT1	Replaces **the CAR** element of **OBJECT1** with a pointer to **OBJECT2** and returns the

OBJECT2)	modified **OBJECT1**.
(RPLACD OBJECT1 OBJECT2)	Replaces **the CDR** element **of OBJECT1** with a pointer to **OBJECT2** and returns the modified **OBJECT1**.
NCONC	Combines the lists "physically" by setting the pointer in the **CDR** field of the last cell of the list that is the first argument to the beginning of the list that is the second argument.
(NSUBSTITUTE NEW OLD LIST TEST)	Replaces with the **NEW** element those **OLD elements of the LIST** list for which the verification flag for the TEST test **is** different from **NIL**.
(NSUBST NEW OLD OBJECT TEST)	Replaces with **NEW** all **OBJECT** subexpressions for which the test flag of the **TEST** test is different from **NIL**.
(DELETE ITEM LIST TEST)	The function destroys all elements of the **LIST** list for which the test flag **TEST** is not **NIL**.
(NCONC LIST1... LISTN)	Returns a list consisting of the elements of lists **LIST1, ...,LISTN** in the same order, by modifying the last **CDR** elements of **LIST1, ..., LISTN**.
(SPLIT LIST)	Splits the list **LIST** into two lists by replacing **the CDR** element of the top-level dotted pair in the middle of **LIST** with **NIL**.
(NREVERSE LIST OBJECT)	Returns the inverted list **LIST** concatenated with the **LISP language object OBJECT**.
(NBUTLAST LIST N)	Returns a list consisting of all elements of LIST **except** the last **N** elements, by replacing the **CDR** element of **the N** th dotted pair counted from the end of **LIST** with **NIL**.

The simplest modifiers that change the physical structure of lists are the **RPLACA** (RePLACe cAr), **RPLACD** (RePLACe cDr) functions, which write new values into the **CAR** and **CDR** fields of a list cell, and the **NCONC** (CONCateNate) function, which "physically" connects lists.

---

*Note: Before reading further, remember that a side effect of calling **RPLACA** or **RPLACD** may be **to change the values of all expressions that use the transformed cell in their internal representation**.*

If **OBJECT1** is neither **NIL** nor a number, then the function **(RPLACA OBJECT1 OBJECT2)** replaces **the CAR** element of **OBJECT1** with a pointer to **OBJECT2** and returns the modified **OBJECT1**.

The result of such a substitution depends on the type of **OBJECT1**.

If **OBJECT1** is a list, the first element of the list is replaced by **OBJECT2**. For example:

```
    $ (SETQ FOO '(A B C))
    (A B C)
    $ (RPLACA FOO D)
    (D B C)
    $ FOO
    (D B C)
```

If **OBJECT1** is a binary tree (dotted pair), then the left branch of the tree is replaced by **OBJECT2**.

If **OBJECT1** is a symbol but not **NIL**, then the value element of the symbol is taken to be **OBJECT2**.

In all cases, a ***modified*** **OBJECT1** is returned.

If **OBJECT1** is neither **NIL** nor a number, then the function **(RPLACD OBJECT1 OBJECT2)** replaces **the CDR** element **of OBJECT1** with a pointer to **OBJECT2** and returns the ***modified*** **OBJECT1**.

The result of such a substitution depends on the type of **OBJECT1**.

If **OBJECT1** is a list, the tail of the list is replaced by **OBJECT2**. For example:

```
  $ (SETQ FOO '(A B C))
  (A B C)
  $ (RPLACD FOO '(D E))
  (A D E)
  $ FOO
  (A D E)
```

If **OBJECT1** is a binary tree, then the right branch of the tree is replaced by **OBJECT2**.

If **OBJECT1** is a symbol but not **NIL**, then the symbol's property list is replaced by **OBJECT2**.

In all cases, a modified OBJECT1 is returned.

Example 1.

```
  $ (RPLACA '(5. 6) 11)
  (11. 6)
  $ (RPLACD '(5. 6) 11)
  (5. 11)
```

Example 2. Let's calculate the values of the following expressions:

```
   1. (RPLACD '(A) B) Result: (A. B)
```

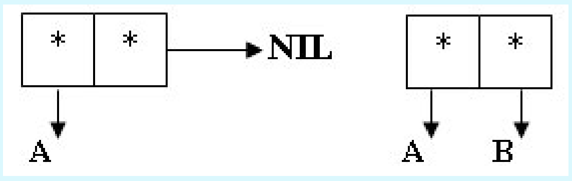


Fig.1. Initial scheme and result

```
  2. (RPLACA '(A) B) Result: (B) or (B. NIL)
```

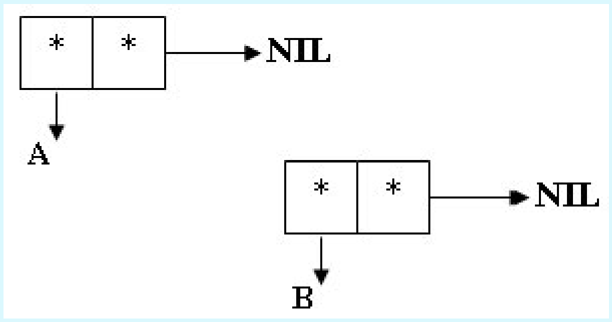


Fig.2. Initial scheme and result

```
  3. (RPLACD '(NIL) NIL) Result: (NIL) or (NIL. NIL)
```

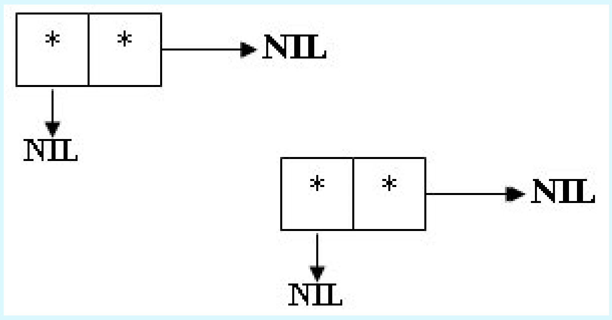


Fig.3. Initial scheme and result

```
4. (RPLACD '(NIL) '(NIL)) Result: (NIL NIL) or (NIL. (NIL. NIL))
```

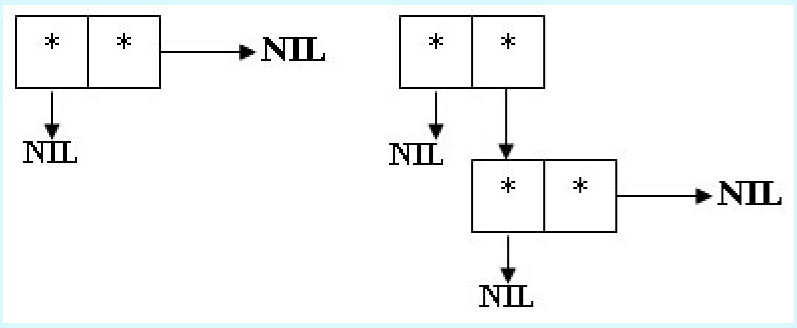


Fig.4. Initial scheme and result

Example 3. Let the value of atom **X** be a list **(A B C)**:

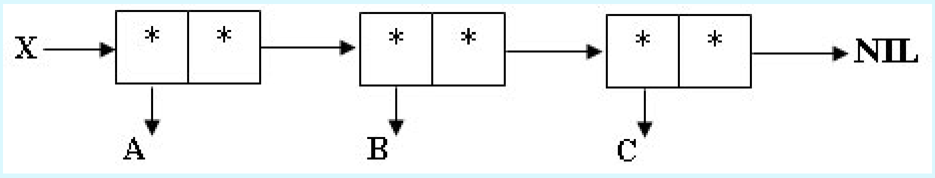


Fig.5. Initial list

After executing **(RPLACD (CDDR X) X)** we get

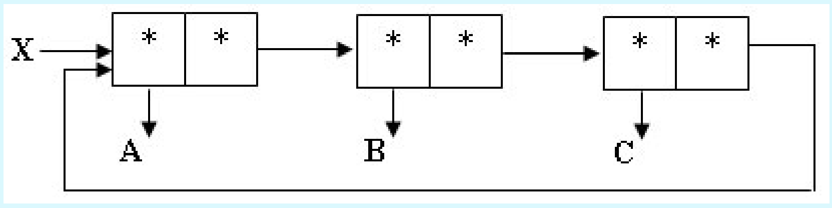


Fig.6. Result of the operation

The values of **X** have no equivalent in the language because they are a cyclic list structure!

After executing **(CONS (CAR (CDDR X)) X)** we get

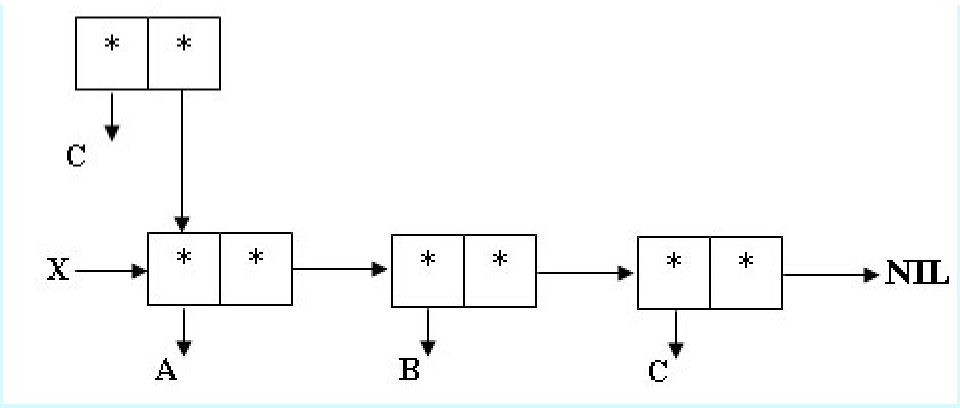


Fig.7. Result of the operation

The NCONC function concatenates lists "physically" by setting the pointer in the **CDR** field of the last cell of the list that is the first argument to the beginning of the list that is the second argument. For example:

```
    $ (SETQ A '(1 2 3))
    (1 2 3)
    $ (SETQ B '(4 5 6))
    (4 5 6)
    $ (SETQ C (NCONC A B))
    (1 2 3 4 5 6)
    $ A
    (1 2 3 4 5 6)
```

If the value of **the FOO** atom is a list, then the function **(NCONC FOO FOO)** will create ***a circular list***, and if an attempt is made to display the value of **FOO** on the display screen, this process will continue indefinitely. For example:

```
    $ (SETQ A '(1 2 3))
    (1 2 3)
    $ (NCONC A A)
    (1 2 3 1 2 3 1 2 3 1 2 3 1 2 3...
```

Pseudo-functions like **RPLACA**, **RPLACD**, and **NCONC** that change the internal structure of lists are useful when you need to make small changes to a large data structure. But sneaky side effects like changing the values of external variables can lead to surprises in parts of the program that "don't know" about the changes. This makes programs difficult to write, document, and maintain, so functions that change structures are rarely used!

---

*Note: The **RPLACA** and **SET** functions are identical. However, **RPLACA** returns its first argument, while **SET** returns its second. Typically, **SET** is used when its first argument is an atom, and **RPLACA** is used when its first argument is a dotted pair.*

Taking into account the above, the code for the SET function can be written as follows:

```
   (DEFUN SET (SYM OBJ)
      ( (SYMBOLP SYM) ( (NULL SYM) OBJ )
                      ( RPLACA SYM OBJ )
                     OBJ)
      ( BREAK (LIST 'SET SYM OBJ) '"Nonsymbolic Argument")
   )
```

In the **muLISP-85** version there are several more modifiers: **NSUBSTITUTE, NSUBSTITUTE-IF, NSUBST, NSUBST-IF, DELETE, DELETE-IF, NREVERSE, NBUTLAST, TCONC, LCONC, SPLIT, SORT, MERGE.**

```
    NSUBSTITUTE [new,old,list,test]    Function
    NSUBSTITUTE-IF [new,test,list] function
```

1. ***By modifying*** the high-level dotted pairs of the **LIST** list, the **(NSUBSTITUTE NEW OLD LIST TEST)** function replaces with the **NEW element those OLD** elements of the **LIST** list for which the verification flag for the **TEST** test is different from **NIL**.

If the test argument is **NIL** or not specified, the **NSUBSTITUTE** function uses **the EQL** test. For example:

```
   $ (NSUBSTITUTE 5 2 '(4 2 (3. 2) 4))
   (4 5 (3. 2) 4)
   $ (NSUBSTITUTE 'CANNIBALS 'NOUN '(NOUN LIKE TO EAT NOUN))
   (CANNIBALS LIKE TO EAT CANNIBALS)
```

The function **(NSUBSTITUTE-IF NEW TEST LIST)** replaces with **NEW** elements those elements of the **LIST** list for which the test verification flag is different from **NIL**.

Here is the function code:

```
   (DEFUN NSUBSTITUTE (NEW OLD LST TEST)
      ( (ATOM LST) LST )
      ( ( (NULL TEST) (SETQ TEST 'EQL) ) )
      ( (FUNCALL TEST OLD (CAR LST))
           (RPLACA LST NEW)
           (NSUBSTITUTE NEW OLD (CDR LST) TEST)
      )
      ( NSUBSTITUTE NEW OLD (CDR LST) TEST )
   )
```

2. When modifying OBJECT dotted pairs, the **(NSUBST NEW OLD OBJECT TEST)** function replaces with **NEW all OBJECT** subexpressions for which the test flag for the **TEST** test is different from **NIL**. If the test argument is **NIL** or not specified, the **NSUBST** function uses **the EQL** test. For example:

```
   $ (NSUBST 5 2 '(4 2 (3. 5 ) 4))
   (4 5 (3. 5 ) 4)
```

The (NSUBST NEW TEST OBJECT) function replaces with **NEW all OBJECT** subexpressions for which the test flag for the **TEST** test is different from **NIL**.

Here is the function code:

```
   (DEFUN NSUBST (NEW OLD OBJ TEST)
      ( ((NULL TEST) (SETQ TEST 'EQL)) )
      ( (FUNCALL TEST OLD OBJ) NEW )
      ( (ATOM OBJ) OBJ )
      ( RPLACA OBJ (NSUBST NEW OLD (CAR OBJ) TEST) )
      ( RPLACA OBJ (NSUBST NEW OLD (CDR OBJ) TEST) )
   )
```

3. The function **(DELETE ITEM LIST TEST)** deletes all elements of the list **LIST** for which the test flag for the **TEST** test is not **NIL**. If the test argument is **NIL** or is not specified, the **DELETE** function uses **the EQL** test. For example:

```
   $ (DELETE '(2 5) '((5 2) (2 5) (2 3)) 'EQUAL)
   ((5 2) (2 3))
```

The (DELETE-IF TEST LIST) function deletes all elements of the **LIST list for which the TEST** test flag is different from **NIL**. For example:

```
   $ (DELETE-IF 'MINUS '(-2 0 7 -0.1 3))
   (0 7 3)
```

Here is the function code:

```
   (DEFUN DELETE (ITEM LST TEST RLST )
      ( ( (NULL TEST) (SETQ TEST 'EQL) ) )
      (LOOP
          ( (ATOM LST) )
          ( (FUNCALL TEST ITEM (CAR LST)) )
          (POP LST)
      )
      ( (ATOM LST) LST )
      ( SETQ RSLT LST )
      (LOOP
          ( (ATOM (CDR LST)) RSLT )
          ( ( (FUNCALL TEST ITEM (CADR LST))
              ( RPLACD LST (CDDR LST) )
            )
          (POP LST) )
      )
   )
```

4. The **(NCONC LIST1... LISTN)** function returns a list consisting of the elements of the lists **LIST1, ..., LISTN** in the same order, by modifying the last **CDR** elements of **LIST1, ..., LISTN**.

Note that the **NCONC** and **APPEND** functions, given the same arguments, will return equivalent lists.

If the value of **FOO** is a list, then **(NCONC FOO FOO)** will create a circular list, and if an attempt is made to display the value of **FOO**, the process will continue indefinitely. For example:

```
   $ (SETQ FOO '(D E F))
   (D E F)
   $ (NCONC '(A B C) FOO '(G H I))
   (A B C D E F G H I)
   $ FOO
   (D E F G H I)
```

Here is the function code:

```
   (DEFUN NCONC (LST1 LST2)
      ( (ATOM LST1) LST2 )
      ( RPLACD (LAST LST1) LST2 )
      LST1
   )
```

5. The **(SPLIT LIST)** function splits the list **LIST** into two lists by replacing **the CDR** element of the top-level dotted pair located ***in the middle*** of the list **LIST** with **NIL**. The **SPLIT** function returns the tail **of the LIST**, starting from the split point.

If a LIST **has** an ***even*** number of elements, then its head and tail will have the same number of elements when split.

If the list has an ***odd*** number of elements, then the head will have one ***more*** element than the tail. For example:

```
   $ (SETQ NAMES '(TJM SUE JOE PAT SAM))
   (TOM SUE JOE PAT SAM)
   $ (SPLIT NAMES)
   (I'M PAT)
   $ NAMES
   (TOM SUE JOE)
```

Here is the function code:

```
  (DEFUN SPLIT (LST)
     ( (ATOM LST) LST )
     (split-aux LST (CDR LST))
  )
  (DEFUN split-aux (HEAD TAIL)
     ( (ATOM TAIL) (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
     (POP TAIL)
     ( (ATOM TAIL) (PROG1 (CDR HEAD) (RPLACD HEAD NIL)) )
     ( split-aux (CDR HEAD) (CDR TAIL) )
  )
```

6. The **(NREVERSE LIST OBJECT)** function returns the inverted list **LIST** concatenated with a LISP object **OBJECT**. The result is similar to that returned by the **(NCONC (NREVERSE LIST) OBJECT)** function, but calling **NREVERSE** with two arguments is more efficient. Note that **NREVERSE** modifies the top-level dotted pairs of **LIST** !

Here is the implementation of the **NREVERSE** function:

```
  (DEFUN NREVERSE (LAMBDA (LST OBJ)
     (COND ( (ATOM LST) OBJ )
           (  T  (NREVERSE (CDR LST) (RPLACD LST OBJ)) )
     )
  ))
```

Test examples:

```
  $ (NREVERSE '(A B C D E))          $ (SETQ FOO '(A B C))
  (E D C B A)                        (A B C)
  $ (NREVERSE '(A B C) '(D E F))     $ (NREVERSE FOO)
  (C B A D E F)                      (C B A)
  $ (NREVERSE '(A B C) 'D)           $ FOO
  (C B A. D)                        A
```

7. If **N** is zero or a positive integer, then the **(NBUTLAST LIST N)** function returns a list consisting of all elements of LIST **except** the last **N** elements, by replacing the **CDR** element **of the N** th dotted pair, counted from the end of LIST, with **NIL**. The absence of the **N** argument identifies **N** with 1. For example:

```
  $ (SETQ FOO '(A B C))    $ (NBUTLAST '(A B C D))
  (A B C)                  (A B C)
  $ (NBUTLAST FOO)         $ (NBUTLAST '(A B C D) 2)
  (A B)                    (A B)
  $ FOO
  (A B)
  $ (NBUTLAST FOO 2)
  NIL
  $ FOO
  (A B)
```

In the next step we will look at ***control structures***.

Step 57.
Control structures. Compound “operators”

In this step we will look at ***the possibility of creating compound operators***.

The PROG1 function transforms a standard functional notation of a **LISP** program into a non-functional notation in the form of a sequence of function calls, reminiscent of a program notation in an imperative programming language.

Syntax of the **PROG1** function:

```
    (PROG1 S-expression1 S-expression2... S-expressionN)
```

The function has a variable number of arguments, which it evaluates sequentially and returns the value of the first one as its value. For example:

```
    $ (PROG1 (SETQ X 2) (SETQ Y (+ X 2)))
    2
    $ And
    4
```

The PROG1 function is often used to avoid introducing temporary variables to store results while evaluating other expressions.

---

*Note: The **muLISP-85** version has the following functions to facilitate support for the imperative programming style: **PROG1, PROGN, CATCH, THROW, UNWIND-PROTECT**.*

---

1. The function **(PROGN FORM1 FORM2 ... FORMN)** evaluates the forms sequentially, starting with **FORM1**, and returns the value of ***the last*** evaluated form.

How the form is calculated depends on the structure of the form list, i.e.:

- if the form **FORM** is ***an atom*** or **(CAR FORM)** is an atom or body **of LAMBDA**, then the function **(EVAL FORM)** is the value of the form **FORM** ;
- if **(CAAR FORM)** is ***an atom*** or body **of LAMBDA**, then **(CAR FORM)** is a predicate and **(CDR FORM)** is the body of an implicit **COND**.

  If the value of the predicate (i.e. **(EVAL (CAR FORM))**) is **NIL**, then the value of the **FORM** form is also **NIL**.

  However, if the value of the predicate is not **NIL**, then all other forms from **PROGN** are removed, and instead the forms in the body of the implicit **COND** are evaluated sequentially.

  If the **COND** body is empty, the **PROGN** function returns the value of the predicate;

- if the **FORM** is none of the above, then the function **(APPLY 'PROGN FORMA)** is evaluated and returned as **the FORM** before continuing to evaluate the remaining forms. This allows for conditional branching and recombination of programs later. For example:

```
    (PROGN (SETQ NUM1 (+ 2 5)) (SETQ NUM2 (* 3 4))
           ( (< NUM1 NUM2) ( (MINUSP NUM1) (* 3 NUM2) )
                             (+ NUM1 NUM2) )
    )
```

The function will return 19.

2. The function **(PROG1 FORM1... FORMN)** evaluates **FORM1**, then the remaining forms using the implicit function **PROGN**, and returns the result of evaluating **FORM1**. For example:

```
$ (SETQ FOO '(A B C D))
(A B C D)
$ (PROG1 (CAR FOO) (SETQ FOO (CDR FOO)))
A
$ FOO
(B C D)
```

3. The **(CATCH LABEL FORM1... FORMN)** function returns the result of calculating the forms using the implicit **PROGN function, if a transition to the LABEL** label does not occur during the calculation. If such a transition occurs, the **CATCH** function returns the value set during the transition. The **LABEL** argument is calculated before calculating the forms.

If **LABEL** is **NIL**, then all transitions that occur during the evaluation of the forms are "caught" by the **CATCH** function.

4. The **(THROW LABEL OBJECT)** function immediately ends the execution of the current function body, assigns the **THROW** variable the value **OBJECT** and transfers control back to the previous **CATCH** function that has **the LABEL** label or the **NIL** label.

The CATCH function returns **an OBJECT** (the value of the **THROW** variable). If the **CATCH** function is not found, control returns to the **muLISP** executable driver.

Note that the **THROW** function is ***a computed function***, so **LABEL** and **OBJECT** are evaluated before the transition occurs. For example:

```
$ (CATCH 'FOO (PRINT 'DOG) (CATCH 'BAR (PRINT 'CAT)
                                       (THROW 'FOO 'COW)
                                       (PRINT 'PIG)
                           )
                           (PRINT 'RAT)
  )
DOG
CAT
COW
```

5. The **(UNWIND-PROTECT FORM1... FORMN)** function evaluates **FORM1** and then the remaining forms using the implicit **PROGN** function, even if the evaluation of **FORM1** ended with a transition.

When the **PROGN** function finishes executing, the **UNWIND-PROTECT** function returns the result of the FORM1 evaluation if it completed normally. Otherwise, the function continues executing the branch if the **FORM1** evaluation ended with a branch.

The UNWIND-PROTECT function is useful when you want to "tidy up" something after **FORM1** has been calculated, regardless of how the calculation was completed.

For example, this function will ensure that after calculating the number "pi" with high accuracy, the accuracy will be restored to its original value, even if the calculation of "pi" was interrupted from the keyboard or a **"Memory full"** error occurred:

```
(PROGN (SETQ OLD-PRECISION (PRECISION))
   (UNWIND-PROTECT (PROGN (PRECISION 100) (COMPUTE-PI))
                   (PRECISION OLD-PRECISION))
)
```

In the next step we will look at ***the organization of cyclic constructions***.

Step 58.
Control structures. Organization of cycles

In this step we will look at ***ways to organize loops***.

In numerical computing there is a tendency to use ***iterative methods*** as much as possible (in numerical analysis the effort is mainly directed to the study and use of iterative algorithms). For "symbolic computing" it is natural to use ***recursive methods***, which are much better suited to handling symbolic information structures. However, it is clear that the distinction between numerical and symbolic computing is to some extent artificial.

LISP is a typical and most widespread language of functional programming, based on the wide use of effective recursion mechanisms in the process of calculations. At the same time, to ensure "ideological" compatibility with imperative programming languages, as well as to increase the efficiency of programs in solving some particular problems, a group of functions was introduced into the language that provide the ability to organize ***iterative*** processing of information.

A typical representative of the group of iterative functions is the **LOOP** function, which in general has the form:

```
(LOOP
    Expr1
    Expr2
 ...
    ExprN
)
```

where any S -expressions or special constructions of the form **(C E1 E2... Ek)** can be used as arguments, where **C,E1,E2, ...,Ek** are **S** -expressions.

In the latter case, the list is used as a point of analysis ***of the loop termination condition***. If the evaluation of **the S** -expression **C** returns the value **NIL**, the iteration process continues, otherwise it is terminated, but first **the S** -expressions **E1, E2, ..., Ek** are evaluated sequentially and the last of the obtained values is returned as the value of the **LOOP** function.

Example 1. The **FACTORIAL** function returns the factorial of the number **NUM** in the **ANS** variable.

```
; ------------------------------------------- ;
; An example of imperative programming style ;
; ------------------------------------------- ;
(DEFUN FACTORIAL (LAMBDA (NUM ANS)
    ( (NOT (> NUM -1)) NIL )
    (SETQ ANS 1)
    (LOOP
        ( (EQ NUM 0) ANS )
        (SETQ ANS (* NUM ANS))
        (SETQ NUM (- NUM 1))
    )
))
; --------------------------------------------- ;
; An example of functional programming style ;
; --------------------------------------------- ;
(DEFUN FACT (LAMBDA (X)
    (COND ( (ZEROP X) 1 )
          (  T  (* X (FACT (- X 1))) )
    )
))
```

Example 2. The **NTH** function returns a list obtained by removing the first **NUM** elements from the list **LST**.

```
; ------------------------------------------- ;
; An example of imperative programming style ;
; ------------------------------------------- ;
(DEFUN NTH (LAMBDA (LST NUM)
    ( (NOT (PLUSP NUM)) LST )
    (LOOP
        (SETQ LST (CDR LST))
        (SETQ NUM (- NUM 1))
        ( (ZEROP NUM) LST)
    )
))
; --------------------------------------------- ;
; An example of functional programming style ;
; --------------------------------------------- ;
(DEFUN NTH_1 (LAMBDA (LST NUM)
   (COND ( (ZEROP NUM) LST )
         (  T  (NTH_1 (CDR LST) (- NUM 1)) )
   )
))
```

Example 3. A predicate that allows one to determine whether a given integer **N** is prime.

```
; ------------------------------------------- ;
; An example of imperative programming style ;
; ------------------------------------------- ;
(DEFUN SIMPLE (LAMBDA (N)
   (SETQ FLAG 1)
   (COND ( (AND (NOT (ODDP N)) (NOT (EQ N 2))) NIL )
         (  T  ( (SETQ I 3)
                 (LOOP
                    ( (< (CAR (DIVIDE N 2)) I) )
                    ( (EQ (REM N I) 0)
                                 (SETQ FLAG 0) )
                    ( SETQ I (+ I 2) )
                 )
                 (COND ( (EQ FLAG 0) NIL )
                       (   T          T  ))) )
   )
))
; --------------------------------------------- ;
; An example of functional programming style ;
; --------------------------------------------- ;
(DEFUN ISPRIM (LAMBDA (N)
   (COND ( (EQ N 1) NIL )
         ( T (ISPR N 2) )
   )
))
; --------------------- ;
(DEFUN ISPR (LAMBDA (N M)
   (COND ( (EQ M N)                T  )
         ( (EQ (REM N M) 0) NIL )
         ( T (ISPR N (+ M 1)) )
   )
))
; --------------------------------------------- ;
; An example of the functional programming style ;
; ("fast" algorithm) ;
; --------------------------------------------- ;
(DEFUN ISPRIM1 (LAMBDA (N))
   (COND ( (EQ N 1)     NIL    )
         (    T    (ISPR1 N 2) )
   )
))
```

```
; ---------------------- ;
(DEFUN ISPR1 (LAMBDA (N M)
   (COND ( (> (* M M) N)   T  )
         ( (EQ (REM N M) 0)   NIL )
         (           T             (ISPR1 N (+ M 1)) )
   )
))
```

*Note: The **muLISP-85** version has the following functions to help support the imperative programming style: **SYSTEM, IF, IDENTITY, COMMENT, RETURN, RESTART, EXECUTE**.*

1. The **(SYSTEM)** function closes all open files, terminates **muLISP** execution, and returns control to the operating system.

If **muLISP-85** is running under **MS-DOS**, and if **INT** is zero or a positive integer less than 256, the **(SYSTEM INT)** function returns **an exit code** (also called ***a return code***) **INT** to the controlling process. The exit code can be queried with the **MS-DOS command "IF ERRORLEVEN"**.

If **INT** is missing, the **muLISP** system returns a return code, which by default is stored in the **muLISP** memory page.

2. The **(RESTART)** function closes all open files, "abandons" the current **muLISP** environment, and initiates a new **muLISP** system. All variable associations, user functions, and property values in the current environment are destroyed.

3. The **(COMMENT COMMENTS)** function "ignores" its arguments and returns **NIL**.

The COMMENT function defines ***a way to include comments directly in a function definition***. Such comments use memory areas in **RAM** memory.

In contrast, comments delimited by comment macro characters are ignored when read into **muLISP**. Although such comments take up space in the source file on disk, they do not use memory areas in **RAM**.

4. The function **(IDENTITY OBJECT)** returns **an OBJECT** without any other actions. The implicit **COND** predicate must be a function application, not just a variable. Therefore, **IDENTITY** can be used to use variables as predicates. For example:

```
$ (IDENTITY NIL)     $ (IDENTITY '(A B C))
NIL                  (A B C)
$ (IDENTITY 'DOG)
DOG
```

Using the **IDENTITY function, we will construct the TEST-VAR** function:

```
(DEFUN TEST-WHERE (WHERE)
   ( (IDENTITY VAR) 'WAR-IS-TRUE) 'WAR-IS-FALSE
)
$ (TEST-VAR NIL)
EXISTS-FALSE
```

Here is the code for the **IDENTITY** function:

```
(DEFUN IDENTITY (OBJ)
   OBJ
)
```

5. If the value of **the PREDICATE** predicate is not **NIL**, then the function **(IF PREDICATE THEN ELSE)** is evaluated and the value of **THEN** is returned. Otherwise, **ELSE** is evaluated and returned, or **NIL** if **ELSE** is missing.

The IF function in **muLISP-85 is similar to the "if-then-else"** construct in imperative programming languages. For example:

```
    $ (IF (EQ (* 2 3) 6) 'TRUE 'FALSE)
    TRUE
    $ (IF (EQ (+ 2 3) 6) 'TRUE 'FALSE)
    FALSE
```

The function code can be written as follows:

```
   (DEFUN IF (NLAMBDA (PREDICATE THEN ELSE)
      ( (EVAL PREDICATE) (EVAL THEN) )
      ( EVAL ELSE )
   ))
```

6. The **(RETURN OBJECT)** function suspends execution of the function containing **RETURN**, clears the stacks, and returns the value of **OBJECT** as its value. For example:

```
   (DEFUN return-test ()
      (PRINT 'DOG) (RETURN 'CAT) (PRINT 'PIG)
   )
   (return-test)
   DOG
   CAT
```

7. The function **(EXECUTE PROGRAM COMMAND-LINE)** suspends **muLISP**, then loads and executes the program **PROGRAM, passing it the command string COMMAND-LINE** as an argument.

When **PROGRAM** terminates, control returns to **muLISP** and the **EXECUTE** function returns the exit code from **PROGRAM**.

If PROGRAM **is not found, the EXECUTE** function returns **NIL**.

The most common use of the **EXECUTE** function is to invoke a secondary command processor. For example, if the command processor file **COMMAND.COM** is located on drive **A:**, then the **muLISP command**

```
   (EXECUTE "A: COMMAND.COM" "/C DIR B:")
```

will show the contents of the file directory on device **B.**

In the next step we will start looking at ***fundamental data types***.

Step 59.
Fundamental data types. Associative lists

In this step we will look at ***association lists***.

Starting with this step, we will describe the fundamental data types in the **muLISP-81** dialect. Note that it lacks such fundamental data structures as real numbers, strings, arrays, sequences, and structures.

An association list (**A** ***-list***, list of pairs) is a fundamental data type that represents a list of dotted pairs of the form: **((A1 . T1) (A2 . T2)... (AN . TN))**

The first element of the pair (**CAR**) is called ***the key***, and the second (**CDR) is called *the data*** associated with the key. Typically, the key is an atom. The data associated with it can be atoms, lists, or some other **LISP** object.

Let us give a formal definition of an **A** -list in Backus-Naur form:

```
<Association list> ::=
        NIL | (<Dotted pair>. <Association list>)
<Dotted pair> ::=
        (<Atom>. <Atom>) | (<Atom>. <Dot Pair>) |
        (<Dot pair>. <Atom>) |
        (<Dotted pair>. <Dotted pair>)
```

Let us give a graphical representation of the **A** -list:

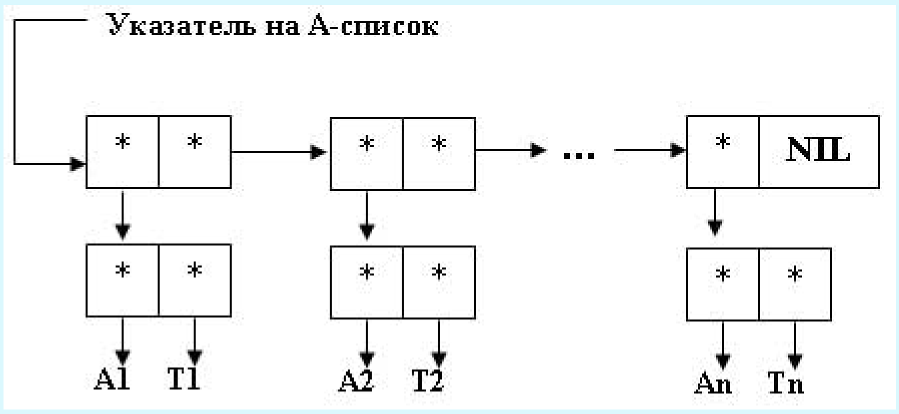


Fig. 1. Graphical representation of **the A**-list

An A-list can be used to combine different types of data components into a single data set.

When working with lists of pairs, you need to be able to build associative lists, search for data by key and update them.

1. Let's construct the **PAIRLIS** function, which can be used to construct **an A** -list from the **KEY** list of keys and the **DATA** list formed from the corresponding data. The third argument is the “old” **A** -list **A-LIST**, ***to the beginning*** of which new pairs are added:

```
    (PAIRLIS KEY DATA A-LIST)
```

Let us define **PAIRLIS** as follows:

```
   (DEFUN PAIRLIS (LAMBDA (KEY DATA A-LIST)
    ;Building an A-list from a list of keys KEY and
   a list of data DATA ;
      ( (NULL KEY)  A-LIST )
      ( (NULL DATA) A-LIST )
      (CONS (CONS (CAR KEY) (CAR DATA))
            (PAIRLIS (CDR KEY) (CDR DATA) A-LIST)
   ))
```

2. The functions **(REKEY ALIST)** and **(REDATA ALIST)** "split" the **A** -list **((X1 . E1)...(Xk . Ek))** and form a separate list of keys **(X1 ... Xk)** and a list of data **(E1 ... Ek)**.

```
   (DEFUN REKEY (LAMBDA (ALIST)
      (COND ( (NULL ALIST) NIL )
            (  T  (CONS (CAR (CAR ALIST))
                        (REKEY (CDR ALIST))) )
      )
   ))
   ; ------------------------- ;
   (DEFUN PLAYED (LAMBDA (ALIST)
      (COND ( (NULL ALIST) NIL )
            (  T  (CONS (CDR (CAR ALIST))
                        (RENDERED (CDR ALIST))) )
      )
   ))
```

3. The function **(ASSOC KEY ALIST)** searches the list of pairs **ALIST** for data corresponding to the key **KEY**, comparing the searched key with the keys of the pairs from left to right.

ASSOC could be defined (simplified) as follows:

```
   (DEFUN ASSOC1 (LAMBDA (KEY ALIST)
    ; Search the list of ALIST pairs of data, ;
   ; corresponding to the key KEY ;
      (COND ( (NULL ALIST) NIL )
            ( (EQ (BRACK ALIST) KEY) (BRACK ALIST) )
            ; The first pair found that
            contains the key KEY is returned;
            (  T  (ASSOC1 KEY (CDR ALIST)) )
      )
   ))
```

Note that **ASSOC** returns a dotted pair as its value, or **NIL** if the **KEY** is not in the association list.

Let's build a function **RASSOC** that searches for a key based on known data **DATA** in the list **ALIST** (**RASSOC** is an abbreviation for the English words "Reverse" and "ASSOCiate"):

```
   (DEFUN RASSOC (LAMBDA (DATA ALIST)
    ; Search the list of ALIST pairs for a key that ;
   ; matches the data DATA ;
      (COND ( (NULL ALIST) NIL )
            ( (EQ (CDAR ALIST) DATA) (CAR ALIST) )
            ; A pair containing the data DATA is returned;
            (  T  (RASSOC DATA (CDR ALIST)) )
      )
   ))
```

4. The association list can be updated and used in stack mode. New pairs are added to it only at the beginning of the list, although the list may already contain data with the same key. This is done by the **ACONS** function:

```
   (DEFUN ACONS (LAMBDA (X Y ALIST)
    ; Adding a pair (X. Y) to the beginning of the ALIST list;
      (CONS (CONS X Y) ALIST)
   ))
```

Since **ASSOC** scans the list from left to right and only gets to the first pair with the searched key, the older pairs are sort of "in the shadow" of the newer ones. The advantage of this is that it is easy to change the links and it is possible to return to the values of the old links. The disadvantage is that the data search slows down proportionally to the number of elements in the list.

5. If the old value is no longer needed, the **ALIST** association list can be changed by physically changing the data associated with the key. This is done using the **RPLACD** pseudo-function that changes the value of the **CDR** field: **(RPLACD (ASSOC KEY ALIST) DATA)**.

The value of the first argument to the call will be a dotted pair whose **CDR** field corresponding to the **KEY** key is then replaced by the **DATA** data from the second argument.

Let's define the pseudo-function **PUTASSOC** (PUT ASSOCiation):

```
(DEFUN PUTASSOC (LAMBDA (KEY DATA ALIST)
 ; Change the data corresponding to the key KEY, ;
; to the data DATA ;
   ( (NULL ALIST) NIL )
   ( (EQ (CAAR ALIST) KEY) (RPLACD (CAR ALIST) DATA) )
   ( (NULL (CDR ALIST))
                (RPLACD ALIST (LIST (CONS KEY DATA))) )
   ; Adding a new pair (KEY. DATA) ;
   ( PUTASSOC KEY DATA (CDR ALIST) )
))
```

The pseudo-function **PUTASSOC** operates on the list not on the logical level, but on the physical level. This means that if we used the old **A**-list in some other value, then this value may change as a result of the side effect of the update due to the action **of PUTASSOC**.

Example: Library of functions for working with associative lists.

```
(DEFUN TEST (LAMBDA NIL
    ; Initialization ;
   (SETQ A '(JAPAN RUSSIA))
   (SETQ B `(TOKYO MOSCOW))
   ; ------------------------------------------ ;
   (PRIN1 "Let's build an A-list from lists A and B: ")
   (PRINT (SETQ ALIST (PAIRLIS A B NIL)))
   ; ----------------------------------------- ;
   (PRIN1 "Let's check the components of the A-list: ")
   (PRINT (REKEY ALIST))
   (PRINT (PLAY ALIST))
   ; ------------------------------------------------- ;
   (PRIN1 "Find data corresponding to the key RUSSIA: ")
   (PRINT (ASSOC1 RUSSIA ALIST))
   (PRIN1 "Find data corresponding to the key JAPAN: ")
   (PRINT (ASSOC1 JAPAN ALIST))
   ; -------------------------------------------------- ;
   (PRIN1 "Find a key that matches the data MOSCOW: ")
   (PRINT (RASSOC MOSCOW ALIST))
   ; -------------------------------------------------- ;
   (PRINT "Appending dotted pair to A-list: ")
   (PRINT (SETQ ALIST (ACONS USA WASHINGTON ALIST)))
   ; ----------------------------------------------------- ;
   (PRINT "Changing data corresponding to key RUSSIA: ")
   (PUTASSOC RUSSIA SPB ALIST)
))
; ------------------------------------- ;
(DEFUN PAIRLIS (LAMBDA (KEY DATA ALIST)
; Construct an A-list from a list of keys KEY and a list of ;
; data DATA by adding new pairs to the existing
list ALIST ;
   ( (NULL KEY)  ALIST )
   ( (NULL DATA) ALIST )
   ( CONS (CONS (CAR KEY) (CAR DATA))
                (PAIRLIS (CDR KEY) (CDR DATA) ALIST) )
))
```

```
; ----------------------------- ;
(DEFUN ASSOC1 (LAMBDA (KEY ALIST)
; Search the list of ALIST pairs for data corresponding to ;
; the key KEY ;
   (COND ( (NULL ALIST) NIL )
         ( (EQ (BRACK ALIST) KEY) (BRACK ALIST) )
         ; 1The first pair found
         containing the key KEY is returned;
         (  T  (ASSOC1 KEY (CDR ALIST)) )
   )
))
; ------------------------------ ;
(DEFUN RASSOC (LAMBDA (DATA ALIST)
; Search the ALIST list of pairs for a key that matches ;
; the data DATA ;
   (COND ( (NULL ALIST) NIL )
         ( (EQ (CDAR ALIST) DATA) (CAR ALIST) )
         ; A pair containing the data DATA is returned;
         (  T  (RASSOC DATA (CDR ALIST)) )
   )
))
; ---------------------------- ;
(DEFUN ACONS (LAMBDA (X Y ALIST)

; Adding a dotted pair (X. Y) to the beginning of the ALIST list ;
   (CONS (CONS X Y) ALIST)
))
; ------------------------------------ ;
(DEFUN PUTASSOC (LAMBDA (KEY DATA ALIST)
; Changing the data corresponding to the key KEY, ;
; to the data DATA ;
   ( (NULL ALIST) NIL )
   ( (EQ (CAAR ALIST) KEY) (RPLACD (CAR ALIST) DATA) )
   ( (NULL (CDR ALIST))
         (RPLACD ALIST (LIST (CONS KEY DATA))) )
   ; Adding a new pair (KEY. DATA) ;
   ( PUTASSOC KEY DATA (CDR ALIST) )
))
; ------------------------ ;
(DEFUN REKEY (LAMBDA (ALIST)
; Recovering a list of keys from a known A-list;
   (COND ( (NULL ALIST) NIL )
         (  T  (CONS (CAR (CAR ALIST))
                     (REKEY (CDR ALIST))) )
   )
))
; ------------------------- ;
(DEFUN PLAYED (LAMBDA (ALIST)
; Recovering a data list from a known A-list;
   (COND ( (NULL ALIST) NIL )
         (  T  (CONS (CDR (CAR ALIST))
                     (RENDERED (CDR ALIST))) )
   )
))
```

*Note: In **muLISP-85, the ASSOC** and **RASSOC** functions are modified as follows.*

***The (ASSOC KEY A-LIST TEST)** function performs **a linear search** in the associative list **A-LIST** for a dotted pair for which, when comparing its **CAR** element with the key **KEY** using the **TEST** test, the value is not equal to **NIL**. If the test argument is **NIL or is not specified, the ASSOC** function uses **the EQL** test.*

*The function **(ASSOC-IF TEST A-LIST)** searches in the associative list **A-LIST** for a dotted pair for which the verification flag of its **CAR** element according to the **TEST** test is not **NIL**.*

*For both functions, the following is true: if a dotted pair satisfying the test is found, that pair is returned; otherwise **NIL** is returned. For example:*

```
$ (SETQ CAPITALS '((USA.WASHINGTON) (FRANCE. PARIS)
(JAPAN. TOKYO)))
((USA.WASHINGTON) (FRANCE. PARIS) (JAPAN. TOKYO))
$ (ASSOC 'FRANCE CAPITALS)
(FRANCE. PARIS)
$ (ASSOC 'AUSTRALIA CAPITALS)
NIL
```

The function can be defined as follows:

```
(DEFUN ASSOC (KEY ALIST TEST)
    ( (ATOM ALIST) NIL )
    ( (ATOM (CAR ALIST)) (ASSOC KEY (CDR ALIST) TEST) )
    ( ( (NULL TEST) (SETQ TEST 'EQL) ) )
    ( (FUNCALL TEST KEY (CAAR ALIST)) (CAR ALIST) )
    ( ASSOC KEY (CDR ALIST) TEST)
)
```

*The function **(RASSOC KEY A-LIST TEST)** performs a linear search in the associative list **A-LIST** for a dotted pair for which, when comparing its **CDR** element with the key **KEY** according to the test **TEST,** the flag is not equal to **NIL**. If the test argument is **NIL** or is not specified, the **RASSOC** function uses **the EQL** test.*

*The function **(RASSOC-IF TEST A-LIST)** searches in the associative list for a pair for which the verification flag of its **CDR** element according to the **TEST** test is not equal to **NIL**.*

*For both functions, the following holds: if a pair that satisfies the test is found, that pair is returned; otherwise, **NIL** is returned. For example:*

```
$ (RASSOC 'PARIS CAPITALS)
(FRANCE. PARIS)
$ (RASSOC 'CANBERRA CAPITALS)
NIL
```

The function can be defined as follows:

```
(DEFUN RASSOC (KEY ALIST TEST)
    ( (ATOM ALIST) NIL )
    ( (ATOM (CAR ALIST)) (RASSOC KEY (CDR ALIST) TEST) )
    ( ( (NULL TEST) (SETQ TEST 'EQL) ) )
    ( (FUNCALL TEST KEY (CDAR ALIST)) (CAR ALIST) )
    ( RASSOC KEY (CDR ALIST) TEST )
)
```

In the next step we will look at ***the property list***.

Step 60.
Fundamental data types. Property lists

In this step we will look at ***property lists***.

We already know that atoms can be endowed with various properties - an atom can be the name of a variable, a constant, a function, or a number. All this is reflected in memory in the form of the so-called ***property list of***

the atom (**P** -list, **Property List**). Among other properties, this list must also contain **the P-name** (this is the external representation of the atom) - a sequence of letters representing it in the program.

The property list may be empty.

An atom together with its property list has the structure shown below (you will find out what **the P-name** of an atom is if you look at remark 3 at the end of this step):

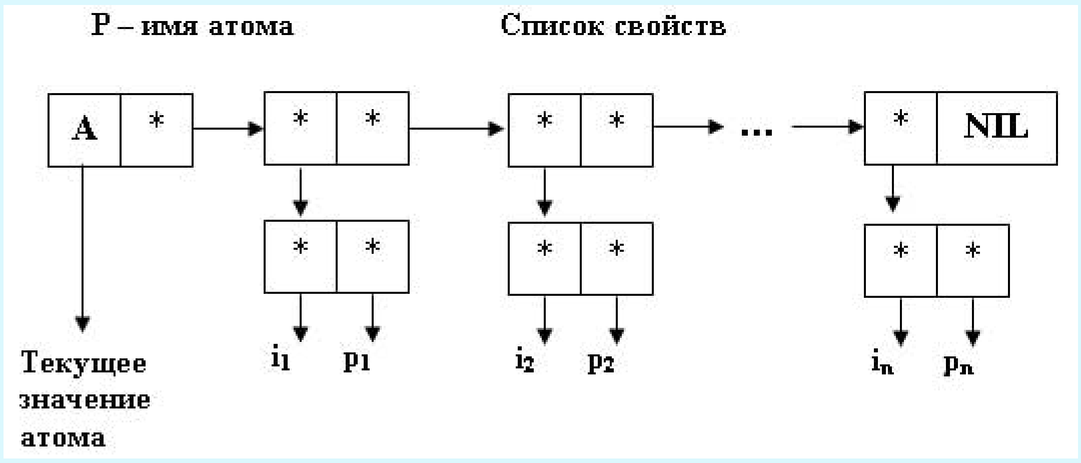


Fig. 1. Atom structure with a list of properties

Access to the property list is always opened through the information cell of the given atom. In the figure, the letter **A** denotes a special address by which information cells can be recognized.

The property list consists of links - two cells in each link. The first cell of each link contains the so-called ***indicator*** **i** $_k$ - the name of the property. More precisely, ***the indicator*** is the address of the information cell of the atom used as the name of the property. The second cell of the link contains ***the address*** **p** $_k$ of the property itself - the defining expression of the function.

The list of atom properties can be updated or deleted as needed, but the programmer must ***provide*** and process the properties of interest to him.

1. ***Assigning a new property*** or changing the value of an existing property is done using the **PUT pseudo-function**

```
   (PUT VARIABLE PROPNAME PROPVAL)
```

Where:

- **VARIABLE** - atom;
- **PROPNAME** - property name;
- **PROPVAL** - the value of the property.

The PUT function returns the value of its third argument, **PROPVAL**. For example:

```
   $ (PUT USA CAPITAL WASHINGTON)
   WASHINGTON
   $ (PUT FRANCE CAPITAL PARIS)
   PARIS
   $ (PUT JAPAN CAPITAL TOKYO)
   TOKYO
```

The side effect is to modify the property list of the **VARIABLE** atom:

- if the property list did not contain a property named **PROPNAME**, then a property with that name and value **PROPVAL** will be added to the list;
- If a property named **PROPNAME** was already present in the property list of the **VARIABLE** atom, the value of that property will be replaced by the new value **PROPVAL**.

Here is the code for this function:

```
 (DEFUN PUT (SYM KEY OBJ)
    ( (NULL (ASSOC KEY (CDR SYM)) )
    ( RPLACD SYM (ACONS KEY OBJ (CDR SYM))) OBJ )
    ( RPLACD (ASSOC KEY (CDR SYM)) OBJ ) OBJ
 )
```

2. You can ***find out the value of a property*** associated with an atom using the **GET** function:

```
  (GET VARIABLE PROPNAME)
```

Where:

- **VARIABLE** - atom;
- **PROPNAME** - the name of the property.

For example:

```
$ (PUT A SVOISTVO 1111)
1111
$ (GET A SVOISTVO)
1111
```

The example below assumes that a **PUT** command was issued.

```
$ (GET FRANCE CAPITAL)
PARIS
$ (GET USA CAPITAL)
WSHINGTON
$ (GET AUSTRIA CAPITAL)
NIL
```

If the **VARIABLE atom does not have the PROPNAME** property, the function returns **NIL**.

Here is the code for this function:

```
 (DEFUN GET (SYM KEY)
    ( (NULL (ASSOC KEY (CDR SYM))) NIL )
    ( CDR (ASSOC KEY (CDR SYM)) )
 )
```

3. ***Deleting a property*** and its value is performed using the **REMPROP** pseudo-function:

```
  (REMPROP VARIABLE PROPNAME)
```

Where:

- **VARIABLE** - atom;
- **PROPNAME** - the name of the property.

The REMPROP function returns the name of the property being removed as its value. If there is no property being removed, **NIL** is returned.

A property can be "deleted" by setting it to NIL:

```
   (PUT VARIABLE PROPNAME NIL)
```

In this case, the property name and the **NIL** value physically remain in the property list.

The example assumes that the above actions with the **PUT** function have been performed:

```
  $ (REMPROP USA CAPITAL)
  WASHINGTON
  $ (GET FRANCE CAPITAL)
  PARIS
  $ (GET USA CAPITAL)
  NIL
```

Here is the code for this function:

```
  (DEFUN REMPROP (SYM KEY)
     ( (ATOM (CDR SYM)) NIL )
     ( (EQUAL (CAADR SYM) KEY)
          (SETQ KEY (CDADR SYM))
          (RPLACD SYM (CDDR SYM))
          KEY )
     ( REMPROP (CDR SYM) KEY )
  )
```

Example 1. Let **X** and **Y** be atoms that have a property **VAL** whose value is a number. The **VALPLUS** function returns the sum of the **VAL** property values of its arguments.

```
  (DEFUN VALPLUS (LAMBDA (X Y VAL)
   ; X and Y are atoms ;
  ; VAL is a property of X and Y atoms ;
     (+ (GET X VAL) (GET Y VAL))
  ))
```

Let's test the given function by performing the following sequence of actions:

```
  $ (PUT X VAL 3)
  3
  $ (PUT AND VALUE 4)
  4
  $ (VALPLUS X Y VAL)
  7
```

*Refine the function definition to include a test of the **VAL** property value using the **NUMBERP** predicate.*

The next example is very important!

Example 2.

```
  $ (SETQ A 0)
  0
  $ (PUT A SVOISTVO1 111)
  111
  $ (PUT A SVOISTVO2 222)
  222
  $ (CAR A)
  0
  $ (CDR A)
  ((SVOISTVO1 . 111) (SVOISTVO2 . 222))
```

Thus, ***calculating the* CDR function *from* the P- *name of the atom allows you to "penetrate" the property list!*** It turns out that the property list is an associative list, where the keys are replaced by properties, and the data associated with the keys are replaced by the values of the corresponding properties.

4. To describe properties that take only two possible values, the LISP language has a ***flag*** mechanism.

A flag is a property without a value. The only thing that can be said about a flag is whether an atom has one or not.

To provide an atom with a flag, there is a function **FLAG**. This function has two arguments. The first is a list of atoms, the second is the name of the flag:

```
    (FLAG ATOM FLAGNAME)
```

Where:

- **ATOM** - atom;
- **FLAGNAME** - the name of the flag that will appear on the atom.

The FLAG function makes **FLAGNAME** the first element in the property list of **ATOM** if it was not already there. For example:

```
   $ (FLAG JOHN MALE)     $ (FLAG SUE FEMALE)
   MALE                   FEMALE
```

Here is the function code:

```
   (DEFUN FLAG (SYM ATTRIB)
      ( (FLAGP SYM ATTRIB) ATTRIB )
      ( RPLACD SYM (CONS ATTRIB (CDR SYM)))
       ATTRIB
   )
```

The function **(REMFLAG ATOM FLAGNAME) "removes" the FLAGNAME** flag from the **ATOM** atom (removes the **FLAGNAME flag from the ATOM** atom's property list) and returns **T**. If the flag is not found, the function returns **NIL**. For example:

```
   $ (FLAG JAN MALE)
   MALE
   $ (FLAG JAN TALL)
   TALL
   $ (REMFLAG JAN MALE)
   T
   $ (FLAGP JAN MALE)
   NIL
   $ (FLAGP JAN TALL)
   (TALL)
```

The function code looks like this:

```
   (DEFUN REMFLAG (SYM ATTRIB)
      ( (ATOM (CDR SYM)) NIL )
      ( (EQUAL ATTRIB (CADR SYM)) (RPLACD SYM (CDDR SYM))
        T )
      ( REMFLAG (CDR SYM) ATTRIB )
   )
```

If **FLAGNAME** is an element of the property list of the atom **ATOM**, then the function **(FLAGP ATOM FLAGNAME)** returns a value other than **NIL** (namely: the property list starting with **FLAGNAME**), otherwise **NIL**. For example:

```
   $ (FLAGP JOHN MALE)
   (MALE)
   $ (FLAGP SUE MALE)
   NIL
```

Here is the function code:

```
   (DEFUN FLAGP (SYM ATTRIB)
      (MEMBER ATTRIB (CDR SYM) 'EQUAL)
   )
```

Example 3.

```
   (DEFUN DEMO_1 (LAMBDA NIL
       ; Assign values to the properties of the atom P1 ;
      (ROAD P1 PROP1 111)
      (PUT P1 PROP2 222)
      (PUT P1 PROP3 333)
      ; Let's look at the property values;
      (PRINT (GET P1 PROP1))
      (PRINT (GET P1 PROP2))
      (PRINT (GET P1 PROP3))
      ; Let's look at the list of properties ;
      (PRINT (CDR P1))
      ; Set a flag named FL at atom P1 ;
      (FLAG P1 FL)
      ; Let's check for the presence of a flag named FL at atom P1 ;
      (PRINT (FLAGP P1 FL))
      ; Let's look at the list of properties ;
      (PRINT (CDR P1))
      ; "Remove" the FL flag from the P1 atom;
      (REMFLAG P1 FL)
      ; Let's check for the presence of a flag named FL u of atom P1 ;
      (PRINT (FLAGP P1 FL))
      ; Let's look at the list of properties ;
      (PRINT (CDR P1))
   ))
```

Results of the function:

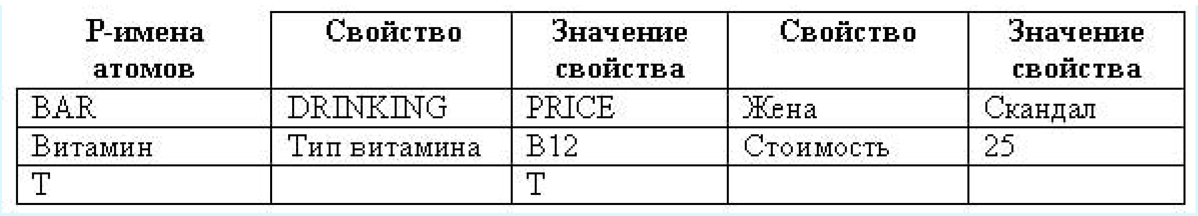
```
   111
   222
   333
   ((PROP3 . 333) (PROP2 . 222) (PROP1 . 111))
   (FL (PROP3 . 333) (PROP2 . 222) (PROP1 . 111))
   (FL (PROP3 . 333) (PROP2 . 222) (PROP1 . 111))
   NIL
   ((PROP3 . 333) (PROP2 . 222) (PROP1 . 111))
   ((PROP3 . 333) (PROP2 . 222) (PROP1 . 111))
```

Property lists serve as useful repositories of information, reflecting knowledge about objects, their properties, and relationships. Most classic large programs rely heavily on the use of property lists.

Data type-specific functions in the form of **LAMBDA** expressions can be stored in property lists of the atoms describing the type. These functions can then be easily retrieved when an argument of the given type appears. This facilitates ***data-driven programming***, a style of programming in which the line between programs and data becomes even more blurred than usual. Data-driven programming is especially appropriate when, at the beginning of a system's construction, it is difficult to foresee the types of objects to be considered.

Notes.

1. *Atoms and their properties can be graphically represented in the form of* ***a table of properties****:*



Р-имена атомов	Свойство	Значение свойства	Свойство	Значение свойства
BAR	DRINKING	PRICE	Жена	Скандал
Витамин	Тип витамина	B12	Стоимость	25
T		T		

Fig.2. Table of properties

It is easy to see that a property table is another way of presenting a list of the properties of an atom. Indeed, look at the corresponding property table lists of the atoms ***BAR*** *and* ***Vitamin****:*

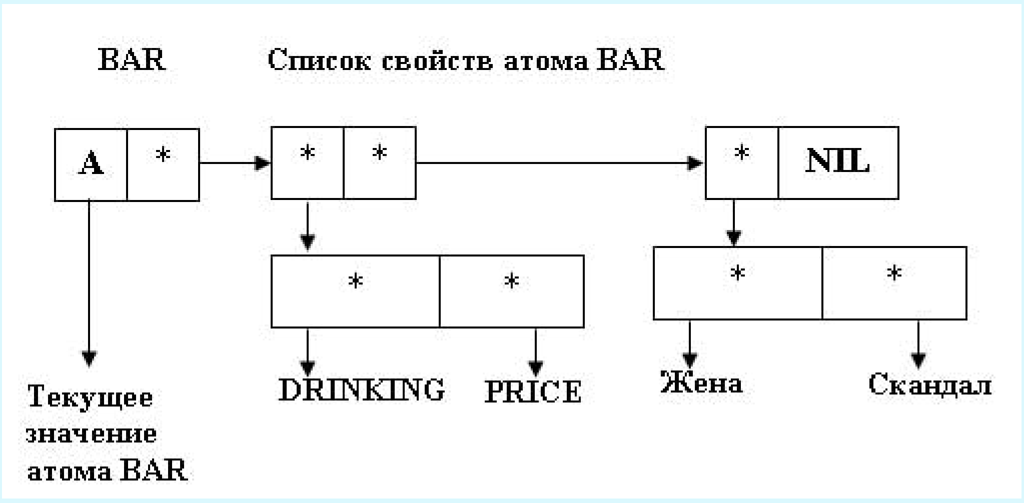


Fig.3. List of properties of the **BAR atom**

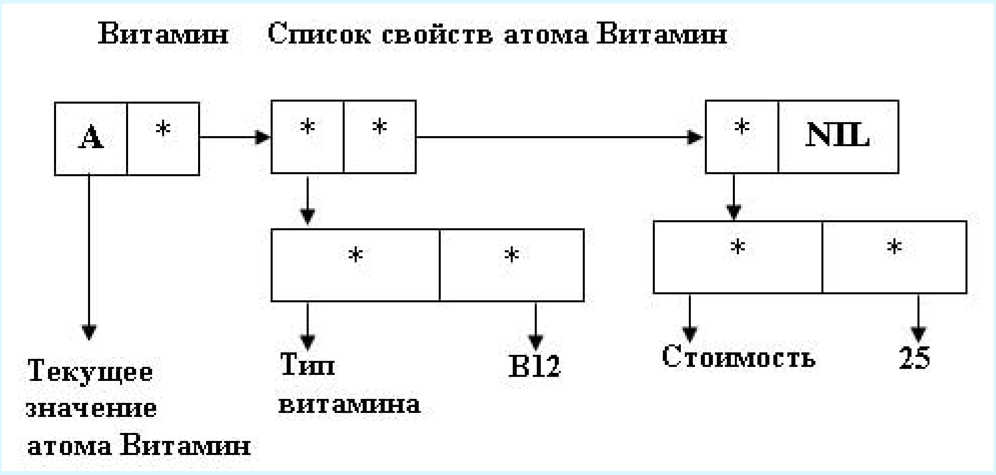


Fig.4. List of properties of ***the Vitamin atom***

Roughly speaking, ***the value of an atom*** *is simply one of the properties that the atom can have. The atoms* ***T*** *and* ***NIL*** *are special atoms in that their values are predetermined to be* ***T*** *and* ***NIL****.*

2. *The introduction of property list atom support into* ***LISP facilitated efficient implementation:***
   - *concepts of* ***frames*** *and* ***frame networks*** *;*
   - *object-oriented programming paradigms.*

The concept ***of a frame*** *is informally defined by M. Minsky as "a data structure for representing a stereotypical situation. Each frame is associated with information of different types. One part of it indicates how the given frame should be used, another - what its implementation can presumably entail, and a third - what should be done if these expectations are not confirmed. A frame can be imagined as a* *network consisting of nodes and connections between them. The "upper levels" of the frame are clearly defined, since they are formed by such concepts that are always true in relation to the supposed situation. At lower levels there are many special vertices-terminals or "cells" that must be filled with characteristic examples or data" [1, p. 7].*

*It should be noted that in artificial intelligence the meaning of the concept of "frame" has been transformed. M. Minsky understood **the frame of an object or phenomenon** as its **minimal description**, which contains all the essential information about this object or phenomenon and has the property that the removal of any part of it from the description leads to the loss of essential information, without which the description of the object or phenomenon cannot be sufficient for their identification.*

*Later, this interpretation of the concept of "frame" changed. Frames came to be understood as descriptions of the form **"Frame name (Set of slots)"**.*

Each slot has a pair of type (!)

```
  (Slot name . Slot value).
```

It is allowed for a slot to be a frame itself. Then the slot values are a set of slots. Constants, variables, any valid expressions in the selected knowledge model, references to other slots and frames, etc. can be used to fill the slots.

***A network** can be understood as any graph structure, but more often it is a graph with some special property (it is accordingly stipulated in the definition of the network) [2, p.109].*

3. *As you have known for a long time, the simplest data types in **muLISP** are **atoms** and **dotted pairs**. The type of data objects can be determined using functions designed to recognize types, for example: **ATOM, NUMBERP, NAME**, etc.*

*Each data object of a particular type consists of a fixed number **of pointers**, which we will henceforth call **elements**. Elements point to other data objects (i.e., contain the addresses of their locations in memory). The set of all data objects forms a connected network of pointers called **a data area**. Elements can point to other objects only within a data area. Therefore, a data area is "closed" in this sense.*

***An atom** is a data object consisting of four pointer elements:*

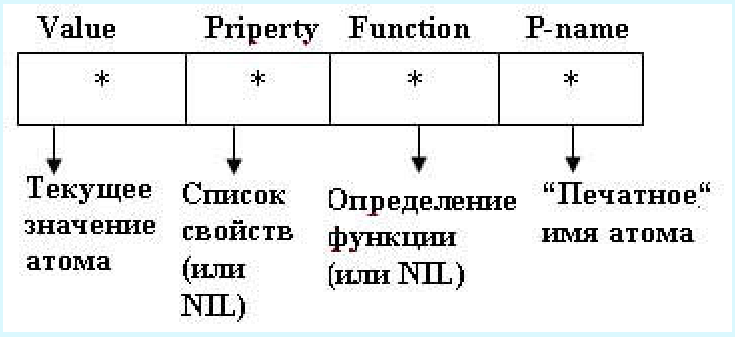


Fig.5. Structure of the atom

Let us describe their purpose:

- ***Value** (pointer to the value of the atom) - this element contains a pointer to the current value of the atom. When an atom is created, its value is taken by itself (its "printable" name), and a reference to it is set in the **Value** element. Such an automatic reference of a symbol to itself is called **an autoreference**. To change the contents of the **Value** element of an atom, functions such* *as **SET, SETQ**, etc. are used. When a function is called, the contents of the **Value element are temporarily redefined** taking into account the actual parameters of the function, and after the execution of the function body, the contents of **the Value** element are restored;*

- ***Property** (property list pointer) - this element contains a pointer to the atom's property list. The list is used and modified using special functions for working with properties (**PUT, GET, REMPROP**) and flags. When an atom is created, **Property** contains **NIL** ;*
- ***Function** (function pointer) - this element contains a pointer to a function definition. When an atom is created, its function element contains **NIL** ;*
- ***P-name** (pointer to the "printable" name of the atom). This element contains a pointer to **the Print** -name of the atom. For the **muLISP-85** dialect, the first 2 bytes contain the length of the atom name, and the rest - the name of the atom itself. Therefore, names are "limited" in size - no more than 65536 characters. As soon as the body **of the Print** -name is read or formed using some string function, a **hashing** algorithm is included to determine whether an atom with the same **Print** -name already exists. If so, the existing atom is used. If not, a new atom with a new **P-name** is created. No two atoms in the system can have the same P-name, moreover, **the P-name** of an atom, once created, cannot be modified.*

In view of the above, it becomes clear that a literal atom can simultaneously name a value and a function, and these possibilities do not interfere with each other. The position of the atom in the expression determines its interpretation. For example:

```
$ (DEFUN LIST1 (LAMBDA (X Y)) (CONS (CONS Y NIL)))
LIST1
$ (SETQ LIST1 A)
A
$ (LIST1 LIST1 B)
(A B)
$ (GETD LIST1)
(LAMBDA (X Y) (CONS (CONS Y NIL))
```

4. *Note that the list of properties can be organized in different ways. We will give the simplest one [3, p.66-67].*

The list has the structure shown below (the meaning of the notations is the same as before):

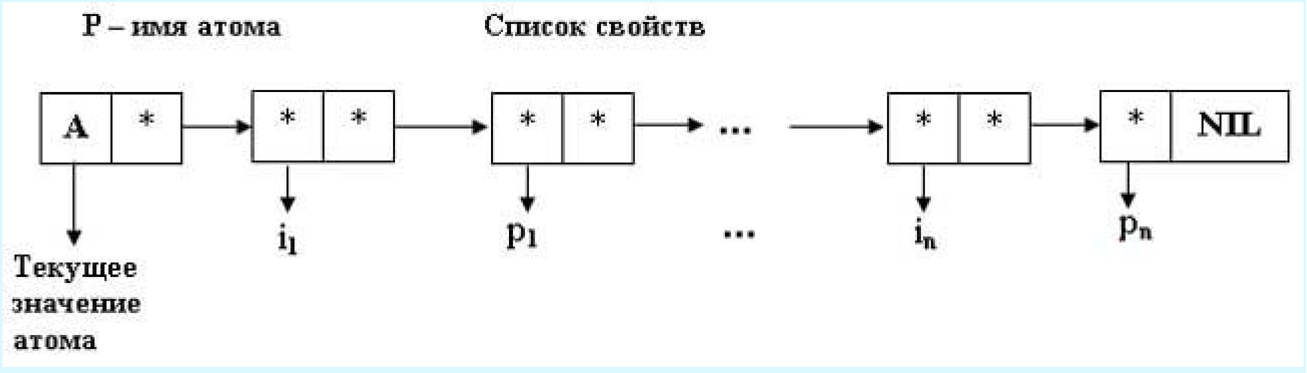


Fig.6. List structure

The described structure of atom property lists is convenient in that the number of properties of an atom is not limited; along with standard properties, any additional properties that may be needed in the program can be assigned to the atom. To study and transform property lists, you can use the same tools that are used to process any other lists.

*The disadvantage of this structure is that property lists take up **a lot of memory** and can be time consuming to scan.*

[1] Minsky M. Frames for knowledge representation. - M.: Energiya, 1979. - 152 p.
[2] Agafonov V.N. Program specification: conceptual means and their organization. - Novosibirsk: Nauka, 1987. - 240 p.
[3] Lavrov S.S., Silagadze G.S. Automatic data processing. The Lisp language and its implementation. - M.: Nauka, 1978. - 176 p.

In the next step we will look at using ***the stack***.

Step 61. Fundamental data types. Stack

In this step we will look at ***working with the stack***.

The PUSH function ***pushes* an S** -expression onto the stack:

```
   (PUSH S-expression Stack)
```

A call **to PUSH** adds an **S -expression to the beginning of the list that is the value of the *Stack*** atom, updating the value of the ***Stack*** atom.

The POP function ***pops*** an element from the stack. Its syntax is:

```
   (POP Stack)
```

Example 1.

```
  $ (SETQ Y (4 5))
  (4 5)
  $ (PUSH (3) Y)
  ((3) 4 5)
  $ (POP AND)
  (3)
  $ And
  (4 5)
```

*Note: The **PUSH** function in the **muLISP-81** version can be defined as follows:*

```
  (DEFUN PUSH (LAMBDA (A STACK)
     (SETQ STACK (CONS A (EVAL STACK)))
  ))
```

Example 2. Copying **NUM** characters of the **ALPHABET** string to the **TOWER** list.

```
  (DEFUN MKTOWER (LAMBDA (NUM ALPHABET TOWER)
     (LOOP
        ( (ZEROP NUM) (REVERSE TOWER) )
        (PUSH (POP ALPHABET) TOWER)
        (SETQ NUM (- NUM 1))
     )
  ))
```

Example 3. Displaying a list on the screen.

```
(DEFUN PRNSTR (LAMBDA (LST)
   (LOOP
      ( (NULL LST) )
      (PRIN1 (POP LST))
   )
))
```

Example 4. The **GENSYM** function is ***an atom constructor*** that returns a new name for an atom of the form **G******, where ******** is a number that increases by one with each new call to the **GENSYM** function.

```
(SETQ GENSYM 0)
(DEFUN GENSYM (LAMBDA (NUM LST)
   (SETQ NUM (- 4 (LENGTH GENSYM)))
   (LOOP
      ( (ZERO NUMBER) )
      ( PUSH 0 LST )
      ( SETQ NUM (- NUM 1)) )
   (PROG1
      (PACK (NCONC (CONS (QUOTE G) LST) (LIST GENSYM)))
      (SETQ GENSYM (+ GENSYM 1)) )
))
```

Example 5. A function that replaces an atom **Y** with a number equal to the depth of occurrence of the atom **Y** in the list **LST** at a given position in the list, for example, if

```
Y = A
LST = ((A B) A (C (A (A D)))),
```

then as a result we get: **((2 B) 1 (C (3 (4 D))))**

```
(DEFUN MAIN (LAMBDA (Y LST)
   (SETQ W (COPY LST LST Y))
   (COPY1 WW)
))
; ---------------------------- ;
(DEFUN COPY (LAMBDA (LST LST1 Y)
; Construction of a list congruent to the list LST and ;
; containing different (!) atoms of the form G*** instead of ;
; occurrences of the atom Y ;
   (COND ( (NULL LST) NIL )
         ( (AND (ATOM LST) (EQ LST Y))  (GENSYM) )
         ( (AND (ATOM LST) (NOT (EQ LST Y))) LST )
         (  T  (CONS (COPY (CAR LST) LST1 Y)
                     (COPY (CDR LST) LST1 Y)
               )
         )
   )
))
; --------------------------- ;
(DEFUN COPY1 (LAMBDA (LST LST1)
; Construction of a list congruent to the list LST and
; containing the "depths of immersion" of G*** atoms in the
; list LST instead of themselves
    (COND ( (NULL LST) NIL )
          ( (AND (ATOM LST)
                 (EQ (CAR (UNPACK LST)) G))
               (POISON LST LST1) )
          ( (AND (ATOM LST)
                 (NOT (EQ (CAR (UNPACK LST)) G)))
               LST )
          (  T  (CONS (COPY1 (CAR LST) LST1)
                      (COPY1 (CDR LST) LST1)) )
    )
))
```

```
   ; ----------- ;
   (SETQ GENSYM 0)
   (DEFUN GENSYM (LAMBDA (NUM LST)
   ; Formation of new atoms ;
      (SETQ NUM (- 4 (LENGTH GENSYM)))
      (LOOP
          ( (ZERO NUMBER) )
          ( PUSH 0 LST )
          ( SETQ NUM (- NUM 1)) )
      (PROG1
          (PACK (NCONC (CONS (QUOTE G) LST) (LIST GENSYM)))
          (SETQ GENSYM (+ GENSYM 1))
      )
   ))
   ; ------------------------ ;
   (DEFUN POISK (LAMBDA (X LST)
   ; Finding the "depth of immersion" of X in the lst LST;
      (COND ( (MEMBER X LST) 1 )
            ( (MEMBER2 X (CAR LST))
                          (+ 1 (POISK X (CAR LST))) )
            (   T    (POISK X (CDR LST)) )
      )
   ))
   ; -------------------------- ;
   (DEFUN MEMBER2 (LAMBDA (X LST)
   ; The MEMBER2 predicate establishes the entry of element X ;
   ; into the multilevel list LST ;
      (COND ( (NULL LST) NIL)
            (  T  (OR (COND ( (ATOM (CAR LST))
                                    (EQ X (CAR LST)) )
                            (  T  (MEMBER2 X (CAR LST)) )
                      )
                      (MEMBER2 X (CDR LST))) )
      )
   ))
```

*Note: We describe the **POP** and **PUSH** functions for the **muLISP-85** version.*

***The (POP SYMBOL)** function returns the "top" **(CAR)** of the stack (list), which is called **SYMBOL**, and replaces the value of **SYMBOL** with the rest of the stack **(CDR)**. For example:*

```
   $ (SETQ STACK-LIST '(A B C D E F))
   (A B C D E F)
   $ (POP STACK-LIST)
   A
   $ (POP STACK-LIST)
   B
   $ (POP STACK-LIST)
   C
   $ STACK-LIST
   (D E F)
```

*If **SYMBOL** is not a symbol, a "Non-symbolic argument" error occurs.*

*If the **SYMBOL** value is not a dotted pair, the **POP** function returns **NIL**.*

*This special function is **the LISP** equivalent of a machine language expression for fetching information from the "top" **of the stack**.*

Let's provide an implementation of the function in the form of a macro:

```
   (DEFMACRO POP (SYM)
      (LIST 'PROG1
            (LIST 'CAR SYM)
            (LIST 'SETQ SYM (LIST 'CDR SYM)))
   )
```

__The (PUSH FORM SYMBOL)__ function evaluates __FORM__, "pushes" the result onto the stack (into __SYMBOL__), and replaces the value of __SYMBOL__ with the expanded stack. The __PUSH__ function returns the expanded stack (list). For example:

```
   $ (SETQ STACK-LIST NIL)
   NIL
   $ (PUSH 'A STACK-LIST)
   (A)
   $ (PUSH 'B STACK-LIST)
   (B A)
   $ (PUSH 'C STACK-LIST)
   (C B A)
   $ STACK-LIST
   (C B A)
```

If __SYMBOL__ is not a symbol, a "Non-symbolic argument" error occurs.

This function is __the LISP__ equivalent of a machine language expression designed to place information at the "top" of the stack.

Let's provide an implementation of the function in the form of a macro:

```
   (DEFMACRO PUSH (OBJ SYM)
      (LIST 'SETQ SYM
           (LIST 'CONS OBJ SYM))
   )
```

From the next step we will start to consider ***working with files***.

Step 62.
Files in muLISP-85. Input Source Functions

In this step, we will look at ***the features that allow you to treat devices as sources of information***.

Input functions and control variables allow programs written in **muLISP** to read and interpret data from ***the current input source*** **(CIS)**.

Input source functions retrieve information related to an input source using input functions. ***The current input source*** **(CIS)** can be created either from the console ***or from the current input file*** **(CIF)**.

Although multiple files can be open for input, only one of the open input files can be a **CIF** file at any given time.

Here is a list of functions that are considered at this step:

Table 1. **Functions for working with the source**

Function	**Purpose**
(RDS drive:name.type)	Looks for file **name.type** on device **drive**.

(INPUTFILE)	Returns the name of the current input file as a character in the form **"drive:name.type"**.
(OPENFILES)	Returns a list of names of currently open input and output files.
(READPTR INTEGER)	Sets the current input file read pointer to **INTEGER** and returns the previous value of the pointer.

Let's describe the functions for working with the current input source.

1. If the file **drive:name.type** has not already been opened for input and/or output, then the function

```
    (RDS drive:name.type)
```

looks for file **name.type** on device **drive**.

If the file is found, the **RDS** function makes it ***the current input file*** **(CIF)**, sets the read pointer to zero, and sets **the RDS control variable to drive:name.type**. This makes the file the ***current input file*** **(CIF)** and ***the current input source*** **(CIS)**.

If the file is not found, the **RDS function does not change the current CIF** input file, and the **RDS** control variable is set to **NIL** ; the console is taken as the current input source (**CIS**).

If the file **drive:name.type** was already open for input and/or output, the function **(RDS drive:name.type)** makes it the current input file (**CIF**) and sets **the RDS** control variable to **drive:name.type**. This makes the file both the current input file (**CIF**) and the current input source (**CIS**). Note that this operation does not provide disk access and that the file's read pointer is not changed.

If **type** is absent, the **RDS** function assigns the file type the name **"LSP"**.

If **drive: is missing, the RDS** function performs the actions described above, except that the file is opened on the currently default device.

If there is an active current input file (**CIF**), then the function call **(RDS)** ***closes*** the current input file (**CIF**) for input (but not for output, if any files are still open for output), i.e. there is no active **CIF**, and sets the variable **RDS** to **NIL** ; the current input source (**CIS**) is assumed to be the console.

Example:

```
   $ (RDS 'D:ANIMAL)
   D:ANIMAL       ; File D:ANIMAL.LSP converted to CIF
```

Normally, the **RDS** function provides source control for input to **muLISP** programs. However, after reading part of a file, you may want to temporarily switch to console input, interrupting the file reading. **The RDS *control variable*** provides this capability.

After using the **RDS** function to open a file and set it as the current input file (**CIF**), the console can become the current input source (**CIS**) without changing **CIF** by setting the **RDS *control variable*** to NIL. A subsequent non- **NIL** value for **RDS** will make the current input file (**CIF**) the current input source (**CIS**) again, and reading of the file will resume where it left off.

2. The **(INPUTFILE)** function returns the name of the current input file (**CIF**) as a character in the form **“drive:name.type”**.

If **drive:name.type** is an open input file, then the function

```
    (INPUTFILE drive:name.type)
```

returns **T**, otherwise returns **NIL**.

If **.type** is missing, the file type is assumed to be **".LSP"**. If **drive:** is missing, the current device is taken.

3. The **(OPENFILES)** function returns a list of the names of currently open input and output files. Each name is a symbol in the form "device:name.type".

4. If **INTEGER** is zero or a positive integer, then the function

```
    (READPTR INTEGER)
```

sets the current input file read pointer to **INTEGER** and returns the pointer's previous value.

If **INTEGER** is neither zero nor a positive integer, the **(READPTR INTEGER)** function returns the current value of the read pointer of the current input file.

The (READPTR 'EOF) function returns the size of the current input file in bytes.

If there is no current input file, the **READPTR** function returns **NIL**.

The file read pointer specifies the location in the file from which the next element will be read. When the **RDS** function opens a file for input, the file read pointer is set to **0**. When an element is read from the file, the file read pointer is automatically incremented.

Note that the **READPRT** function operates on the current input file, but not on the current input source, so the current input source (**CIS**) must be ***the console***.

A good example of the use of the **READPTR** function is the **muLISP *learning system***. This system reads and displays lesson texts on the screen using the **READFILE** function. Since the system must provide the ability to view the previous screen, the **READFILE** function "pushes" the read pointers marking each screen onto the **PRTLST** stack. If the student wants to see the previous screen, the function is called **(READPTR (POP PRTLST))** and the read pointer is pushed back to the beginning of the screen, allowing it to be read and displayed again.

In the next step we will list ***the input functions***.

Step 63.
Files in muLISP-85. Input Functions

In this step we will look at ***functions that read from the input source***.

Input functions read information from ***the current input source*** (**CIS**) and return the **CIS** input.

If file is the current input source (**CIS**) **and if an attempt is made to read data beyond the end of the file, an End-of-file** error interrupt occurs.

Here is a list of functions that are considered at this step:

Table 1. **Input functions**

Function	**Purpose**
(READ-CHAR)	Reads the next element from the current input source (**CIS**) and returns the character whose P-name consists of that character.
(UNREAD-CHAR)	Restores the last element read from the current input source (**CIS**) to the "top" of the current input source and returns **NIL**.
(PEEK-CHAR)	Reads the next element from the current input stream (**CIS**).

(CLEAR-INPUT)	"Clears" the linear editing buffer.
(READ-LINE)	Reads elements from the current input stream until **<ENTER>** is encountered, and returns a character whose P-name consists of all elements read except **<ENTER>**.
(LISTEN)	Returns **T** if the current input stream contains elements available for input, otherwise **NIL**.
(READ)	Reads a single integer expression from the current input stream and returns the equivalent linked list.
(COPY-CHAR-TYPE CHAR1 CHAR2 FLAG)	Copies a **CHAR2** element into the current reading area of a **CHAR1** element.
(WAR)	Reads an element from the current input source and returns the corresponding **muLISP** atom.
(SET-BREAK-CHARS LIST FLAG)	Makes the elements of **LIST** list break characters.

Let us describe the input functions in the **muLISP-85** system.

1. The **(READ-CHAR)** function reads the next element from the current input source (**CIS**) and returns a character whose P-name consists of this character. For ease of programming, the character returned by the **READ-CHAR** function is also assigned to the **RATOM** variable. Note that the **READ-CHAR** function always returns a character even if the element read is a decimal number.

If **PEEK-FLAG** is not **NIL**, then the function

```
    (READ-CHAR PEEK-FLAG)
```

also reads a single element, returning the corresponding character, and assigns it to the variable **RATOM**. However, the element read by **READ-CHAR** remains at the "top" of the input stream, which is used by **READ** and **RATOM** (but not by **READ-CHAR**). This makes it possible to use **READ-CHAR** to "look ahead" past one element, and then use **READ** or **RATOM** to read that element as part of the next input stream.

2. The **(UNREAD-CHAR)** function restores the last element read from the current input source (**CIS**) to the "top" of the current input source and returns **NIL**. Since only the last element read can be restored, calling the function again without performing the read operation will have no effect.

3. The **(PEEK-CHAR)** function reads the next element from the current input stream (**CIS**).

The (PEEK-CHAR T) function reads elements from until a "non-empty" element is encountered.

If **SYMBOL** is a symbol, the **(PEEK-CHAR SYMBOL)** function reads elements from the current input stream until it encounters an element equal to the first character in **the P**-name **SYMBOL**.

In all cases, the **PEEK-CHAR** function restores the last element read from the current input stream to the "top" **of the CIS** and returns the character whose **P**-name consists of that element. For programming convenience, the character returned by **PEEK-CHAR** is also assigned to the **RATOM** variable.

The PEEK-CHAR function always returns a character, even if the last element is a number.

4. If the console is the current input stream, and input from the console is performed using the linear editing method, then the **(CLEAR-INPUT)** function "clears" the linear editing buffer. If the console is the current input stream, and input from the console is performed using the primitive ("rough") input method, then the **CLEAR-INPUT** function clears the primitive ("rough") input buffer.

In any case, the **CLEAR-INPUT** function returns **NIL**.

5. The **(READ-LINE)** function reads elements from the current input stream until **<ENTER>** is encountered and returns a character whose **P**-name consists of all elements read except **<ENTER>**.

The READ-LINE function returns a string exactly as it is, without discarding whitespace or comments.

6. If the current input stream contains elements available for input, then the **(LISTEN)** function returns **T**, otherwise **NIL**.

The LISTEN function is used to determine the readiness state of elements in the current input stream.

If the current input stream is the ***console***, this function can be used to check whether the element is accessible to the user.

If the current input stream is a source file, this function can be used to check whether the end of the file has been reached.

7. The **(READ)** function reads a single integer expression from the current input stream and returns the equivalent linked list. Well-formed expressions using either list comprehensions, dot comprehensions, or combinations of both are valid input to the **READ** function.

Whitespace characters are used only to delimit read characters and are otherwise ignored by the **READ** function.

Delimiters can be considered as part of the characters read by the **READ** function.

A simple delimiter character defines the boundaries of the element immediately following it.

A compound delimiter character defines the boundaries of an element between itself and the next compound delimiter character.

The simple delimiter symbol is the **"\"** symbol, and the composite delimiter symbol is the **"|"** symbol.

The simple delimiter symbol can be used to include simple and compound delimiters in elements whose boundaries are defined by compound delimiters.

8. If **FLAG** is not equal to **NIL**, then the function

```
    (COPY-CHAR-TYPE CHAR1 CHAR2 FLAG)
```

copies a **CHAR2** element into the current reading area of a **CHAR1** element.

If **FLAG is NIL** or absent, the function copies the CHAR2 element **into** the default read range of the **CHAR1** element.

If **CHAR2** is a macro symbol, then the pointer to its macro definition is also copied into **CHAR1**.

If the copying was successful, the function returns **T**, otherwise **NIL**.

9. The **(RATOM)** function reads an element from the current input source and returns the corresponding **muLISP** atom. For programming convenience, the atom called by the **RATOM** function is defined as the value of the **RATOM** variable.

An element is a character string delimited by either spaces or break characters. The **RATOM** function returns a number if the element begins with a decimal number less than the current base number. Otherwise, the **RATOM** function returns a character. Numeric elements end with any character that is not a base number digit.

Whitespace characters are intended only to delimit elements and are not returned as atoms **by the RATOM function.**

By default, the following characters are considered whitespace:

```
   <space>, <enter>, <linefeed>, <page> or <tab>.
```

Break characters also delimit elements, but unlike whitespace characters, they are returned by the **RATOM** function as characters that are part of the elements.

The break symbols are:

```
   ! " # $ % & ' ( ) * +, -. / : < = > ? @ [ \ ] ^ _ ` { | } ~
```

The RATOM function recognizes simple and compound delimiter characters as a way to include spaces and/or break characters as part of stream elements. For example:

```
   $ (WAR)
   DOG
   DOG
   $ (WAR)
   123CAT
   123
```

10. If **FLAG is NIL** or absent, then the function

```
   (SET-BREAK-CHARS LIST FLAG)
```

makes the elements of **LIST** list break characters.

If **FLAG** is not **NIL**, the **SET-BREAK-CHARS** function adds the characters from **LIST** to the current set of break characters.

The function **(GET-BREAK-CHARS)** returns a list of the current set of break characters.

In the next step we will look at ***the macro symbols used in the input functions***.

Step 64.
Input functions. Macro symbols

In this step we will look at ***the macro symbols used in organizing input***.

Macro symbols call their macro definition function as they are read by the **READ** function. They can not only shorten frequently used code segments, but also modify the files used in **LISP** to make them more convenient to use manually.

Macro symbols are divided into:

- limiting,
- non-restrictive and
- comment macro symbols.

Delimiting and non-delimiting macro symbols include objects in the input stream. ***Comment macro symbols*** do not include objects in the input stream.

Delimiting macro symbols and comments delimit the characters read by the **READ** function. Non-delimiting macro symbols do not delimit the characters read by the **READ** function and can therefore be recognized as macro symbols by using spaces.

First, let's note the functions that allow you to work with ***limiting macro symbols***.

1. If **FLAG is NIL** or absent, then the function

```
   (SET-MACRO-CHAR CHARACTER DEFINITION FLAG)
```

compiles **DEFINITION** to **D** code and creates **CHARACTER** in the form of a delimiting macro symbol.

If **FLAG** is the **COMMENT** symbol, **SET-MACRO-CHAR** creates **a CHARACTER** in the form of a comment macro character. Otherwise, the function creates **a CHARACTER** in the form of a nonrestrictive macro character.

2. If **FLAG is NIL** or absent, then the function

```
   (GET-MACRO-CHAR CHARACTER FLAG)
```

decompiles and returns the definition of the macro symbol associated with **CHARACTER**.

If **FLAG** is not **NIL**, then the **GET-MACRO-CHAR** function returns **LAMBDA** if **CHARACTER** is a macro character. In either case, the function returns **NIL** if **CHARACTER** is not a macro character.

Now let's describe ***the limiting macro symbols***:

- **"("** (***left parenthesis***) - ***the left parenthesis macro*** reads expressions until it encounters a matching right parenthesis, right square bracket, or period.

  If ***the right parenthesis*** is encountered, the macro returns a list of the expressions read.

  If ***a right square bracket*** is encountered, the macro skips the bracket so that it can be read again, and returns the list of expressions read. This makes it possible to treat the right square bracket as ***a "super parenthesis"*** that closes all left parentheses for the **READ** function above it.

  If ***a period*** is encountered, the macro reads the expression following the period and checks it to see if the non-whitespace expression is either a right parenthesis or a right square bracket. If so, the macro returns a list of expressions read before the period was encountered that are not related to the expression read after the period. If not, a "Syntax Error" trap occurs.

- **")"** (***right parenthesis***), **"]"** (***right square bracket***) - ***the left parenthesis macro*** recognizes the right parenthesis and right square bracket as list delimiters.

  The right parenthesis and right square bracket macros generate a "Syntax Error" trap if the **READ** function encounters "too" many right parentheses or square brackets.

  The code for the described macros is given below:

```
  (SET-MACRO-CHAR '\)
                  '(LAMBDA () (BREAK '\) "Syntax Error") ))
  (SET-MACRO-CHAR '\]
                  '(LAMBDA () (BREAK '\] "Syntax Error") ))
```

- **","** (***comma***) - the comma macro generates a "Syntax Error" interrupt if it was read directly by the **READ** function.

- **“'”** (***simple quote, apostrophe***) - the simple quote macro reads the expression following the quote and returns a list of two elements **- the QUOTE** symbol and the expression read. The simple quote makes it easy to enter constants into a **LISP** expression. For example:

```
$ '(SUE TOM ANN)
(SUE TOM ANN)
```

Here is the macro code:

```
(SET-MACRO-CHAR (QUOTE (LAMBDA ()
    (LIST (QUOTE QUOTE) (READ))))
)
```

- **"** (***double quote***) - The double quote macro reads an element of the input stream until it encounters a double quote character, and returns a character whose P-name consists of the characters read. Simple delimiters can be used to include double quotes and simple delimiters in the stream. For example:

```
$ "Yom said, \"I like Lisp\""
Tom said, "I like Lisp"
```

Here is the code for the described macro:

```
(SET-MACRO-CHAR '\"
                 '(LAMBDA (CHAR LST)
                      (LOOP
                          (SETQ CHAR (READ-CHAR))
                          ( (EQ CHAR '\")
                               (PACK (NREVERSE LST))
                          )
                          ( ( (EQ CHAR '\\)
                                 (PUSH (READ-CHAR) LST))
                            (PUSH CHAR LST))))
)
```

- **“;”** (***semicolon***) - the semicolon macro reads elements until it encounters **<ENTER>**. Since in this case objects are not included in the input stream, it is convenient to use a semicolon to limit comments. For example:

```
$ (LIST 'DOG 'CAT ; a comment
    'COW)
(DOG CAT COW)
```

The macro code looks like this:

```
(SET-MACRO-CHAR '\" '(LAMBDA ()
                          (LOOP
                              ( (EQ (READ-CHAR)
                                    (ASCII 13 )))))
                      'COMMENT)
```

In the next step we will look at ***the output functions***.

Step 65.
Files in muLISP-85. Output functions

In this step we will look at ***the information output functions***.

Output functions pass output data to ***the current output stream* (COS)**.

If the current output stream (**COS**) is a disk file and there is not enough disk space to accommodate the output, an attempt is made to set the **"Disk Full"** flag to **NIL. If this attempt is not honored by application programs, a "Disk Full"** error interrupt occurs. When the interrupt occurs, the **WRS** control variable is set to **NIL**, the console is assumed to be the current output stream, but the file remains open.

Here is a list of functions that are considered at this step:

Table 1. **Output functions**

Function	**Purpose**
(WRS FileName)	Switches the current output stream to the file **FileName** (if no extension is specified, the default extension.**LSP** is used)
(PRIN1 OBJECT)	Sends the character representation **of OBJECT** to the current output stream and returns **OBJECT**.
(PRINC OBJECT)	Identical to **PRIN1**, except that **P** names containing special characters are not restricted by delimiting characters, and the value of the control variable ***PRINT-ESCAPE*** does not matter.
(PRINT OBJECT)	Sends the character representation **of OBJECT** to the current output stream (**COS**) using the **PRIN1** function, delimits the string, and then returns **OBJECT**.
(PRIVITY)	Outputs **N ASCII** newline characters to the current output stream and returns **NIL** if **N** is zero or a positive integer.
(FRESH-LINE)	Returns **NIL** if we are already at the beginning of the string.
(SPACES N)	Sends **N ASCII** blank characters (spaces) to the current output stream (**COS**) and returns the number of characters sent after the last newline has been sent.
(WRITE-STRING SYMBOL) or (WRITE-LINE SYMBOL)	Writes the elements of **the P** name **SYMBOL** to the current output stream and returns **SYMBOL**.
(WRITE-BYTE N)	Writes **N** bytes to the current output stream (**COS**) and returns **N**.
(LINELENGTH N)	Specifies the length of a line to print to the system printer in a file, so that lines are automatically limited to **N** characters.

Let's tell you about the output functions in the **muLISP-85** system.

1. The **(WRS FileName)** function switches the current output stream to the file **FileName** (if the extension is not specified, the default extension.**LSP** is used)

The (WRS) function switches the current output stream to the standard device (***display***).

Examples:

```
   $ (WRS MY.lib)
          ; Switches the current output stream to the file "MY.LIB"
   $ (SETQ WRS)
         ; Temporarily switches output to standard
         ; output stream (usually to the screen)
   $ (SETQ T)
         ; Connect output to current output file
```

2. The **(PRIN1 OBJECT)** function sends the character representation **of OBJECT** to the current output stream and returns **OBJECT**.

The PRIN1 function prints characters using their P-names. The ***PRINT-ESCAPE*** variable controls the use of delimiters around **the P-names**, i.e., spaces, macros, or delimiters. **PRIN1** prints numbers according to the current number system. The ***PRINT-POINT*** control variable controls the maximum number of decimal places for displaying fractional numbers on the display screen. The **PRIN1** function prints dot pairs using their list images where possible and dot pair images where necessary.

3. The function **(PRINC OBJECT)** is identical to the function **PRIN1**, except that **P** names containing special characters are not limited by delimiting characters, and the value of the control variable ***PRINT-ESCAPE*** does not matter.

The function code looks like this:

```
(DEFUN PRINC (OBJ *PRINT-ESCAPE*)
   (SETQ *PRINT-ESCAPE* T)
   (PRIN1 OBJ)
)
```

4. The **(PRINT OBJECT)** function transfers the character representation **of OBJECT** to the current output stream (**COS**) using the **PRIN1** function, delimits the string, and then returns **OBJECT**.

The function code looks like this:

```
(DEFUN PRINT (OBJ)
   (PRIN1 OBJ) (TERPRI) OBJ
)
```

5. If **N** is zero or a positive integer, then **(TERPRI N)** writes **N ASCII** newline characters to the current output stream and returns **NIL**. If **N** is absent, its value is assumed to be 1.

The function code can be written as follows:

```
(DEFUN TERPRI (N)
   ((AND (INTEGERP N) (>=N 0))
         (LOOP
             ((ZEROP N) NIL)
             ((WRITE-BYTE 13)
             ((WRITE-BYTE 10)
             (DECQ N) ) )
   (PRIVITY 1)
)
```

6. If we are already at the beginning of the line, then the function **(FRESH-LINE)** returns **NIL**, otherwise the function passes the new line to the current output stream (**COS**) and returns **T**.

7. If **N** is zero or a positive integer, then the function **(SPACES N)** sends **N ASCII** blank characters (spaces) to the current output stream (**COS**) and returns the number of characters sent after the last newline has been sent. Any missing **N** is replaced by 1. For example:

```
$ (SPACES 10)
10   ; The cursor has moved to the right along the line by 10 positions
```

8. If **SYMBOL** is a symbol, then the **(WRITE-STRING SYMBOL)** and **(WRITE-LINE SYMBOL)** functions write the elements of **the P-name SYMBOL** to the current output stream and return **SYMBOL**.

The WRITE-LINE function writes a newline element after passing **the P-name**. If **SYMBOL** is not a symbol, both functions return **NIL**.

9. If **N** is an integer between 0 and 255 inclusive, then the **(WRITE-BYTE N)** function writes **N** bytes to the current output stream (**COS**) and returns **N**. For example:

```
    $ (WRITE-BYTE 65)
    65
```

10. If **N** is a positive integer, then the **(LINELENGTH N)** function determines the length of a line for output by the system printer in a file, so that lines are automatically limited to **N** characters.
The **LINELENGTH** function returns the previous length of the line. If **N** is a non-positive integer or is not specified, then the function returns the current length of the line. The default line length is 79.

In the next step we will look at ***the variables that control the output of information***.

Step 66.
Files in muLISP-85. Variables for controlling output

In this step we will look at some variables that ***will help control the output***.

Output control variables are used to enhance **muLISP** programs' control over output symbols. They are most useful here as **OFF (NIL)** or **ON** (not **NIL)** switches.

1. If the control variable **ECHO** is not **NIL** and the disk file is a dotted pair, then characters sent to the file are also sent to the console.

   If **ECHO is NIL**, then characters are not echoed to the console.

2. **The LINELENGTH** control variable can be used to override the automatic line length limit.

   If **LINELENGTH** is not **NIL**, the output lines are automatically limited as determined by the **LINELENGTH** function.

   If **LENGTH is NIL**, then all automatic line length limits are removed. This is especially useful in cases where line length limits are undesirable, such as for the **muSTAR** screen editor or when transferring very large amounts of data to a disk file.

3. If the control variable ***PRINT-DOWNCASE*** is not **NIL** and the symbol has **a P-name** that consists of uppercase letters, spaces, macros, delimiters, or begins with a decimal digit, then the symbol's P-name **is *delimited*** when passed to the current output stream by **the PRINT** and **PRIN1** functions.

   If ***PRINT-DOWNCASE*** is **NIL**, then **the P-names** of such symbols are not delimited.

   Delimiting **P** symbol names with delimiter characters is necessary for programs that generate **muLISP** source files.

4. If the value of the control variable ***PRINT-BASE*** is an integer in the range 2-36 inclusive, then this integer is taken as the base of the number system when calculating the numbers passed to the current

output stream (**COS**). Otherwise, the numbers passed to the current output stream are defined as decimal (i.e., the base of the number system is ten). For example:

```
    $ (SETQ TEN 10)
    10
    $ (SETQ *PRINT-BASE* 16)
    10
    $ TEN
    0A       ; Ten in hexadecimal notation
    $ (SETQ *PRINT-BASE* 2)
    10
    $ TEN
    1010     ; Ten in binary form records
    $ (SETQ *PRINT-BASE* 10)
    10
    $ TEN
    10       ; Ten in decimal notation
```

5. If the control variable *PRINT-POINT* is a non-negative integer, then fractional numbers are displayed on the screen using decimal notation, with ***PRINT-POINT*** specifying the number of digits following the decimal point. Otherwise, fractional numbers are written using a slash (numerator, "/", and denominator).

   Note that the value of the ***PRINT-POINT*** control variable only affects the output of numbers to the screen, but does not affect their precision. In addition, numbers can be written using either decimal notation or slashes, regardless of the value of ***PRINT-POINT***.

   The default value of ***PRINT-POINT* is 7. For example:**

```
    $ (SETQ P1 3.1416)                $ (SETQ *PRINT-POINT* 2)
    3.1416                            2
    $ (SETQ *PRINT-POINT* NIL)        $ P1
    NIL                               3.14
    $ P1                              $ (SETQ *PRINT-POINT* 7)
    3927/1250                         7
                                      $P1
                                      3.1416
```

6. If the control variable ***PRINT-ECHO*** is not **NIL**, then characters sent to the console are also sent to the system ***printer*. If *PRINT-ECHO*** is **NIL**, then no printing is performed. Examples:

```
    $ (SETQ *PRINTER-ECHO* T)
                  ; printer output enabled
    $ (SETQ *PRINTER-ECHO* NIL)
                 ; printer output disabled
```

7. If the control variable ***PRINT-ESCAPE*** is not **NIL**, and a symbol has **a P-name** consisting of uppercase letters, spaces, or macros (or delimiters), or beginning with a decimal digit, then that symbol's P-name is delimited when passed to the current output stream (**COS**) by **the PRINT** and **PRIN1** functions. If ***PRINT-ESCAPE*** is **NIL**, then **the P-name**s of such symbols are not delimited.

Delimiting **the P**-names of such symbols using delimiter characters is necessary for programs that generate **muLISP** source files.

From the next step we will start to look at ***the implementation of encapsulated data types***.

Step 67.
Implementing Encapsulated Data Types (General Information)

From this step we will begin to look at ***creating and using data structures***.

"If you decide that your program is not fast enough, the first thing to do is to make sure that you are solving the problem using the best algorithms and data representations. Replacing a primitive or inadequate algorithm with a more appropriate one can speed up your program by an order of magnitude or more. So if you spend a few days studying the famous books of Knuth [1-3] or Sedgewick [4] (in the hope of finding an algorithm that you would hardly have thought of on your own), rest assured that you have made a profitable investment of your precious time.

Similarly, moving from an "obvious" but simple data structure (such as a linked list) to a more complex one (such as a binary tree) can yield results that will more than pay off your efforts to improve the program."

Ray Duncan. "PC Magazine", 1,1992, p.102

An abstract data type (ADT) is a type that is defined not by the method of constructing values of this type from simpler ones, but by a set of ***operations*** on them and ***axioms*** that these values and operations must satisfy.

This term appeared in 1974 in the article by B. Liskov and S. Zilles [5], devoted to the principles of the **CLU** programming language they were developing. The term quickly spread among system programmers and programming theorists and became not only popular with them, but so fashionable that they began to call concepts very far from what its authors were talking about. The adjective "abstract" can be misleading (especially in mathematics), since abstract types are no more abstract than many other variants of the concept of type, to which this adjective is not "applied". In fact, we are talking primarily about increasing the level of abstraction in relation to imperative programming languages.

The original intuitive idea behind the concept of ***an ADT*** is to group into a single concept several operations (actions) and a set (one or more) of objects to which they apply, and to protect (hide) in the language and the corresponding programming system the internal representation of **the ADT** and the actions not explicitly specified in the definition of ***the ADT***.

In programming languages, ***an ADT*** is a construct (cluster, module, class) consisting of two parts:

- ***appearance*** (interface, coupling, specification), containing the name of the defined ***ADT***, names of operations indicating the types of their arguments and values, etc.;
- ***description of operations and objects*** with which they operate, by means of a programming language. We will call this description concrete, in contrast to a more abstract description by means of a higher level (a concrete description is also called an implementation or representation of **an ADT**). Due to protection (hiding), only the names listed in the externality are accessible, i.e. can be used by other program components external to ***the ADT***.

In some experimental languages, for example in ***Alphard*** [6, p.149], the language construct corresponding to ***the ADT*** contains, in addition to the two mentioned, two more parts:

- ***abstract description*** by means of a higher level, including non-procedural ones;
- ***a description of the correctness of a representation*** that specifies in what sense a concrete description correctly represents an abstract description.

An ADT in a programming language that provides protection (hiding) of the representation, but does not provide an abstract description, is called ***an encapsulated data type (EDT). This term is, of course, closer to the essence of the matter, since "encapsulate" literally means "to place in a capsule, to protect with a shell". In implemented programming languages, only an EDT*** or an even weaker form ***of ADT*** is used- without protection.

An algebraic approach can be applied to the formal recording of abstract data types [7, p.11-12]. It is manifested in the fact that a set of operations on data is specified. Data are defined as structures generated by these operations. The use of expressive means of algebra is associated with the possibility of applying mathematical methods to data, in particular with the prospects for applying the theory of associative and commutative groups to data.

If ***ADT*** is perceived by many programmers with tension and internal resistance, while mathematicians see and create algebraic systems almost at every step, then this is partly explained by the infancy of programming compared to mathematics.

V.N.Agafonov [6]

As an example, let's consider the "list" data type. As operations on lists, we will take the list creation operation **CONS**, the list decomposition operations **CAR** and **CDR**, the **NULL** operation, which makes it possible to determine whether a list is empty or not, and the **NIL** operation, which specifies an empty list.

These operations are written as follows:

```
DATA_TYPE LIST (X) IS
   OPERATIONS:
      NIL : --> LIST(X)
      CONS: X x LIST(X) --> LIST(X)
      CAR : LIST(X) --> X
      CDR : LIST(X) --> LIST(X)
      NULL: LIST(X) --> BOOLEAN
END LIST
```

The symbol **X** in the expressions above denotes the data type for each non-empty element of the list. The list data type is called a polymorphic data type because it contains variables for types such as **X**.

If these definitions alone are used, the relationships between the operations CONS, **CAR,** and **CDR**, as well as the form of the expression **NULL**, remain unclear. Therefore, in addition to specifying the operations, expressions (usually called axioms) are written that these operations must satisfy. There are three list axioms:

```
DATA_TYPE LIST(X) IS
   AXIOMS:
      VAR AND: X,
          L: LIST(X)
      CONS (CAR (L), CDR (L)) = L
      NULL (NIL) = TRUE
      NULL (CONS (X,L)) = FALSE
END LIST
```

With the help of the operations and axioms thus defined, an abstract data type is formally defined: "list". The creation of concrete lists is carried out in such a way as to satisfy these rules.

A data type is ***complete*** if it provides enough operations so that all the user-required operations on objects can be done with acceptable efficiency. There is no strict definition of completeness, although there are limits to how few operations a type can have and still be acceptable.

The completeness of a type depends on the context of use. If a type is intended to be used in a limited context (such as a single program), then sufficient operations for that context should be provided. If a type is intended for general use, it is desirable to have a rich set of operations [8, p.98].

Data abstraction operations are divided into 4 classes [8, p.97].

1. Primitive constructors.

These operations create objects of the corresponding type without using any objects as arguments (for example, creating an empty list). Usually, primitive constructors create only some objects, not all. Other objects are created by constructors or modifiers.

2. Constructors.

These operations take objects of their corresponding type as arguments and create other objects of the same type.

3. Modifiers.

These operations modify objects of their corresponding type (for example, insert and delete operations).

4. Observers (selectors).

These operations take objects of the corresponding type as arguments and return a result of another type. They are used to obtain information about objects. Sometimes observers are combined with constructors or modifiers.

In general, a data abstraction must have operations of at least three of the four classes discussed earlier. It must have primitive constructors, observers, and either constructors (if it is immutable) or modifiers (if it is mutable) [8, p.98].

---

(1) Knuth D. The Art of Computer Programming. V.1: Basic Algorithms. - M.: Mir, 1976. - 736 p.
(2) Knuth D. The Art of Computer Programming. V.2: Seminumerical Algorithms. - M.: Mir, 1977. - 724 p.
(3) Knuth D. The Art of Computer Programming. V.3: Sorting and Searching. - M.: Mir, 1978. - 844 p.
(4) Sedgewick B. Algorithms. Addison-Wesley, Reading, Mass., 1983.
(5) Liskov B., Zilles S. Programming with abstract data types. - SIGPLAN Notices, 1974, v.9, N4, p.50-59.
(6) Agafonov V.N. Program Specification: Conceptual Tools and Their Organization. - Novosibirsk: Nauka, 1987. - 240 p.
(7) Futi K., Suzuki N. Programming languages and circuit design of VLSI. - M.: Mir, 1988. - 224 p.
(8) Liskov B., Gateg J. Using abstractions and specifications in program development. - M.: Mir, 1989. - 424 p.

In the next step we will look at ***creating and using arrays***.

Step 68.
Implementing Encapsulated Data Types. Arrays

In this step we will look at ***creating and using arrays***.

For technological reasons, the architecture of most computers is designed so that program variables are arranged in a linear ("one-dimensional") order. This has far-reaching consequences concerning common programming "habits" [1, pp. 242-243].

If you need a large number of variables for objects of the same type, it is recommended to use ***an array***, i.e. a finite sequence of indexed variables.

Indexes are usually integers. In this case, to specify the range of indexes, it is sufficient to specify the lower and upper bounds. An array can have dimensions of 1, 2, ... ***One-dimensional arrays***, or ***vectors***, form an important special class of arrays.

Example 1. Library for working with arrays. Representation of arrays using **LISP** lists.

```
(DEFUN NTH (LAMBDA (N LST)
 ; The function returns the N-th element of the list LST ;
   (COND ( (EQ N 1) (CAR LST) )
         (    T     (NTH (- N 1) (CDR LST) ) )
   )
))
; -------------------------- ;
(DEFUN INSERT (LAMBDA (X N LST)
;The function inserts element X at the N-th position ;
;into the list LST ;
   (COND ( (NULL LST) (CONS X LST) )
         ( (EQ N 1) (CONS X LST) )
         (  T  (CONS (CAR LST)
                     (INSERT X (- N 1)
                             (CDR LST))) )
   )
))
; ------------------------ ;
(DEFUN DELETE (LAMBDA (N LST)
; The function removes the N-th element from the list LST ;
   (COND ( (EQ N 1) (CDR LST) )
         (  T  (CONS (CAR LST)
               (DELETE (- N 1) (CDR LST))) )
   )
))
;Let's give an example of using library functions:
(DEFUN SWAP (LAMBDA (N M LST)
; Exchange the values of the N-th and M-th elements of the list LST ;
   (INSERT (NTH N LST) M
           (DELETE M (INSERT (NTH M LST) N
                             (DELETE N LST))))
))
```

To test this library, you can exchange the values of **the N-th** and **M**-th elements of the list **LST**:

```
  $ (SWAP 2 3 '(1 2 3 4 5))
  1 3 2 4 5
```

Example 2. We will implement in the **LISP** language a function of the **APL** ***programming language*** called ***restructuring*** (denoted by **@**), the action of which we will analyze using examples:

```
1) Team: 2 3 @ 4 7 8 2 4 6
   Result of execution:
  4 7 8
  2 4 6
2) Team: 4 2 @ 7 8 4
   Result of execution:
  7 8
  4 7
  8 4
  7 8
3) Team: 3 4 @ 1 2 3 4 5 6 7 8 9 10 11 12 13 14
   Result of execution:
  1  2  3  4
  5  6  7  8
  9 10 11 12
```

The numbers to the left of the operation sign determine the structure of the resulting matrix: the number of rows and the number of columns. The numbers to the right of the operation sign are used to construct the array, they are ordered by rows.

Two comments are appropriate here.

First, if you don't have enough elements to build the array, **APL** goes back to the beginning of the "heap" of data on the right side and starts picking out element by element from it again.

Secondly, if, on the contrary, we have too many elements, then exactly as many as needed are selected, the rest are ignored.

```
(DEFUN R (LAMBDA (NC LST)
   (COND ( (<; (LENGTH LST) (* N C))
                    (R N C (APPEND LST LST)) )
         (   T   (COND ( (ZEROP N) NIL )
                       (    T    (CONS (CARN C LST)
                                       (R (- N 1)
                                          C
                                          (CDRN C LST))))
                 )
         )
     )
))
; ----------------------- ;
(DEFUN CARN (LAMBDA (N LST)
   (COND ( (ZEROP N) NIL )
         (  T  (CONS (CAR LST)
                     (CARN (- N 1) (CDR LST))) )
   )
))
; ----------------------- ;
(DEFUN CDRN (LAMBDA (N LST)
   (COND ( (ZEROP N) LST )
         (   T  (CDRN (- N 1) (CDR LST)) )
   )
))
```

Test example:

```
$ (R 2 3 '(4 7 8 2 4 6))
((4 7 8) (2 4 6))
```

Note: In addition to the arrays with numeric indices that we are accustomed to, you can also work with hash arrays.

***Hash arrays** are related to a regular one-dimensional array and an associative list. If a regular array can be used to associate **LISP** objects with integers (indices), and an associative list can be used to associate symbols that are keys, then a hash array can be used to associate two arbitrary **LISP** objects (atoms, lists, strings, etc.).*

***Working with a hash array is similar to working with a regular (one-dimensional) array, with the only difference being that a pointer to a LISP** object is used as an index.*

*The advantage of hash arrays is the speed of calculations. Searching for data corresponding to a key, for example, in an association list, involves sequentially viewing the keys until the desired key is found. Instead, in a hash array, the storage location corresponding to the key is calculated directly from the type of key using a special **hash function**. However, if association lists are small (contain from 4 (!) to 100 elements for different versions of the **LISP** language), they can be more efficient than hash arrays, which require significant time for initialization (**Have A Large Initial Overhead**).*

*Thus, if you use associative lists in your program, replacing them with hash arrays (**Hash Tables**) can **speed up** execution.*

*You can read about hash arrays in the **Perl** programming language in Step 10.*

In the next step we will look at ***working with strings***.

Step 69.
Implementing Encapsulated Data Types. Strings

In this step we will look at ***examples of working with strings***.

First, note that ***a string*** for us is **the P**-name of a string atom. This is a very important remark!

However, the muLISP-81 version lacks functions that allow you to operate with **P**-names of string atoms. Nevertheless, using the **PACK** and **UNPACK** functions, you can reduce working with **P**-names to operations with lists.

Function

```
   (UNPACK A)
```

"splits" the name of atom **A** into a list of atoms whose names are composed of the symbols of the name of atom **A**. For example:

```
  $ (UNPACK ABC)
  (A B C)
```

Function

```
   (PACK List)
```

performs the inverse operation of the **UNPACK** function. For example:

```
  $ (PACK (A B C))
  ABC
```

Example 1.

```
  (DEFUN ATOMCAR (LAMBDA (X)
   ; The function returns the first character of the atom name X ;
     (CAR (UNPACK X))
  ))
  ; ---------------------- ;
  (DEFUN ATOMCDR (LAMBDA (X)
  ; The function returns the name of the atom X without its first character ;
     (PACK (CDR (UNPACK X)))
  ))
```

Example 2.

```
  (DEFUN NUM-STRING (LAMBDA (X)
   ; Convert a non-negative integer to a literal atom ;
     (PACK (UNPACK X))
  ))
```

Example 3.

```
(DEFUN STRING-NUM (LAMBDA (X)
 ; Convert a list of digits to a list containing the corresponding ;
; single-digit numbers ;
   (COND ( (NULL X) NIL )
         (  T  (CONS (POSITION (CAR X)
                               (UNPACK 123456789))
                     (STRING-NUM (CDR X))
               )
         )
   )
))
; ---------------------------- ;
(DEFUN  POSITION (LAMBDA (X LST)
; The POSITION function returns the position of the atom X in a ;
; single-level list LST (the first element has ;
; number 1). If the element is not in the list, the function ;
; returns 0 ;
   (COND ( (NULL LST)       0 )
         ( (EQ X (CAR LST)) 1 )
         ( (MEMBER X LST) (+ 1
                               (POSITION X (CDR LST)))
         )
         ( T  0 )
   )
))
```

Example 4.

```
(DEFUN LONGMULT (LAMBDA (X Y)
; "Long" multiplication of real numbers: ;
; X and Y are atoms representing real numbers. ;
; For example, the number 567.098 corresponds to the representation ;
; A567#098, where A is any letter, the symbol # indicates ;
; the position of the decimal point in the number. ;
; The result is returned in the same form! ;
   (PACK
     (REVERSE
       (INSERT #
           (+
              (+    (-
                       (LENGTH (CDR (UNPACK X)))
                       (POSITION # (CDR (UNPACK X))))
                    (-
                       (LENGTH (CDR (UNPACK Y)))
                       (POSITION # (CDR (UNPACK Y))))
              ) 1
           )
         (REVERSE
           (UNPACK
             (*
                (NUMBER
                  (STRING-NUM
                     (DELETE
                         (POSITION # (CDR (UNPACK X)))
                         (CDR (UNPACK X)))))
                  (NUMBER
                     (STRING-NUM
                        (DELETE
                            (POSITION # (CDR (UNPACK Y)))
                            (CDR (UNPACK Y)))))
             )
           )
```

```
          )
        )
      )
    )
))
; ------------------------- ;
(DEFUN NUM-STRING (LAMBDA (X)
; Convert a non-negative integer ;
; to a literal atom ;
   (PACK (UNPACK X))
))
; ------------------------- ;
(DEFUN STRING-NUM (LAMBDA (X)
; Translate a list of digits into a list containing ;
; the corresponding single-digit numbers ;
   (COND ( (NULL X) NIL )
         (  T  (CONS (POSITION (CAR X)
                               (UNPACK 123456789))
                     (STRING-NUM (CDR X))
               )
         )
   )
))
; ----------------------- ;
(DEFUN NUMBER (LAMBDA (LST)
; Given a numeric list LST containing single-digit ;
; numbers. "Build" an integer from the elements of the given ;
; list. ;
   (COND ( (NULL LST) 0 )
         (  T  (+  (* (CAR LST)
                            (STAGE 10
                            (- (LENGTH LST) 1))
                    )
                     (NUMBER (CDR LST))
               )
         )
   )
))
; ----------------------- ;
(DEFUN STEPEN (LAMBDA (X A)
; Raising an integer X to a non-negative integer ;
; power of A ;
   (COND ( (ZEROP A) 1 )
         ( (ZEROP (- A 1)) X )
         (  T  (* (STEPEN X (- A 1)) X) )
   )
))
; --------------------------- ;
(DEFUN INSERT (LAMBDA (X N LST)
; The function inserts element X at the N-th position ;
; into the list LST ;
   (COND ( (NULL LST) (CONS X LST) )
         ( (EQ N 1) (CONS X LST) )
         (  T  (CONS (CAR LST)
                     (INSERT X
                             (- N 1) (CDR LST)))
         )
   )
))
; ------------------------- ;
(DEFUN DELETE (LAMBDA (N LST)
; The function removes the N-th element from the list LST;
   (COND ( (EQ N 1) (CDR LST) )
         (  T  (CONS (CAR LST)
                     (DELETE (- N 1)
                                    (CDR LST)) )
   )
))
; ----------------------------- ;
(DEFUN POSITION (LAMBDA (X LST)
; The POSITION function returns the position of the atom X in ;
; the single-level list LST (the first element has ;
; number 1). If the element is not in the list, the function ;
; returns 0 ;
   (COND ( (NULL LST)       0 )
         ( (EQ X (CAR LST)) 1 )
         ( (MEMBER X LST) (+ 1
                                    (POSITION X (CDR LST))) )
         ( T  0 )
   )
))
```

*Note: The **PACK** function places the created atom into the **OBLIST** list !*

In the next step we will look at ***working with strings in* muLISP-85**.

Step 70.
Implementing Encapsulated Data Types. Strings in muLISP-85

In this step we will look at ***the string handling functions in* muLISP-85**.

The muLISP-85 version has about a dozen more functions for working with strings, for example: **PACK*, CHAR, STRING=, STRING<, STRING>, STRING<=, STRING>=, STRING/=, STRING-UPCASE, STRING-DOWNCASE, PRINT-LENGTH**.

Here is a list of functions that are considered at this step:

Table 1. **String functions in muLISP-85**

Function	**Purpose**
(UNPACK ATOM)	Returns a list of characters, where **the P**-names of each character consist of characters in the printed ATOM representation.
(PACK LIST)	Returns a symbol whose **P**-name consists of the concatenated **P**-names of the atoms in **LIST**.
(PACK* ATOM1... ATOMN)	Returns a symbol whose **P**-name consists of the concatenated **P**-names **ATOM1, ..., ATOMN**.
(CHAR ATOM N)	Returns the literal atom whose **P**-name is **the N-th** character **of the P**-name **ATOM**.
(STRING ATOM1 ATOM2)	Returns **T** if **the P**-name **ATOM1** is lexicographically equal to **the P**-name **ATOM2**, otherwise **NIL**.
(STRING< ATOM1 ATOM2)	Returns the position number of the first character starting from which P-names do not match.
(STRING-UPCASE ATOM)	Returns an atom whose **P**-name is the same as **the P**-name **of ATOM**, except that all lowercase characters are converted to uppercase.
(STRING-DOWNCASE ATOM)	Returns an atom whose P-name is the same as **the P**-name **of ATOM**, except that all uppercase letters are converted to lowercase.
(PRINT-LENGTH ATOM)	Returns the number of characters required to print **ATOM**, based on the current value of the ***PRINT-ESCAPE*** and ***PRINT-BASE*** system variables.
(FINDSTRING ATOM1 ATOM2 N)	Returns the position number of the first occurrence of **the P-name ATOM1** in **the P-name ATOM2**, counting from 0.

(SUBSTRING ATOM N M)	Returns an atom whose P-name consists of the characters of **the P-name ATOM**, starting from **the N-th** through **the M-th** characters, with characters counted starting from 0.

Let us describe their purpose.

1. The **(UNPACK ATOM)** function returns a list of characters, where **the P**-names of each character consist of the characters in the printed representation of ATOM. If ATOM is not an atom, the **UNPACK** function returns **NIL**. For example:

```
    $ (UNPACK 'ABCDE)
    (A B C D E)
    $ (SETQ FOO -216)
    -216
    $ (UNPACK FOO)
    (- \2 \1 \6)
```

String functions called with numeric arguments automatically create a character string equivalent to the numeric value based on the current number system. It is important to note that the **P** name of the number ***changes*** with the current number system. For example:

```
    $ (SETQ *PRINT-BASE* 16)
    10
    $ (UNPACK FOO)
    (- \0 D \8)
    $ (SETQ *PRINT-BASE* 10)
    10
```

2. The **(PACK LIST)** function returns a symbol whose **P**-name consists of concatenated P-names of atoms in **LIST**. The current number system is used to determine **the P**-names of numbers. Note that the **PACK** function always returns a symbol, even if **the P**-name consists only of single-digit numbers. For example:

```
    $ (PACK '(A B C))
    ABC
    $ (PACK '(3 A 5))
    |3A5|
```

3. The **(PACK* ATOM1 ... ATOMN)** function returns a symbol whose **P**-name consists of the concatenated **P**-names **ATOM1, ..., ATOMN**. The **PACK*** function is a narrower version of the **PACK** function, since it works not with a list of atoms, but with an arbitrary number of atoms. For example:

```
    $ (PACK* 'A 'B 'C)
    ABC
    $ (PACK* 3 'A 5)
    |3A5|
```

The function code is quite simple:

```
    (DEFUN PACK* LST
       (PACK LST)
    )
```

4. If **ATOM** is either a literal or a numeric atom, and **N** is a non-negative integer, then **(CHAR ATOM N)** returns the literal atom whose **P**-name is **the N-th** character of **the P**-name of **ATOM**, counted from 0. **CHAR** returns **NIL** if **N** is neither zero nor a positive integer, or if **the P**-name of **ATOM** contains fewer than **N** characters. For example:

```
    $ (CHAR 'ABCDEFG 3)
    D
    $ (CHAR 5432 0)
    \5
```

The code for this function is simple:

```
    (DEFUN CHAR (ATM N)
       ( (ATOM ATM) (NTH N (UNPACK ATM)) )
    )
```

5. If **the P-name ATOM1** is lexicographically equal to **the P-name ATOM2**, then the function **(STRING ATOM1 ATOM2)** returns **T**, otherwise **NIL**.

If the **FLAG** flag is not **NIL**, the **(STRING ATOM1 ATOM2 FLAG)** function performs a similar comparison, but without distinguishing between uppercase and lowercase letters. For example:

```
    $ (STRING= 'FAST 'FASTER)     $ (STRING= 'Fast 'FAST T)
    NIL                           T
    $ (STRING= 100 |100|)
    T
```

6. If **the P-name ATOM1** is lexicographically less than **the P-name ATOM2**, then the function **(STRING< ATOM1 ATOM2)** returns the position number of the first character starting from which **the P-name**s do not coincide; otherwise, it returns **NIL**.

If **FLAG** is not **NIL**, the **(STRING< ATOM1 ATOM2 FLAG)** function performs a similar comparison, but without distinguishing between uppercase and lowercase letters. Functions

```
     (STRING>  ATOM1 ATOM2 FLAG)
     (STRING<= ATOM1 ATOM2 FLAG)
     (STRING>= ATOM1 ATOM2 FLAG)
     (STRING/= ATOM1 ATOM2 FLAG)
```

also perform a lexicographic comparison and return either **NIL** or the position number of the first non-matching character in **the P-name**s. The **STRING>** function compares P-names for >, **STRING<=** for <=, **STRING>=** for >=, and **STRING/=** for <>. For example:

```
    $ (STRING< 'DOG 'CAT)       $ (STRING<= 'DOG 'DOG)
    NIL                         3
    $ (STRING< 'CAT 'DOG)       $ (STRING/= 'DOG 'DOG)
    0                           NIL
    $ (STRING< 'DOG 'DOGGY)
    3
```

7. The **(STRING-UPCASE ATOM)** function returns an atom whose P-name is the same as **the P-name of ATOM**, except that all lowercase characters are converted to uppercase. If **ATOM** is not an atom, the **STRING-UPCASE** function returns NIL. For example:

```
    $ (STRING-UPCASE "Lisp is recursive")
    |LISP IS RECURSIVE|
```

8. The **(STRING-DOWNCASE ATOM)** function returns an atom whose P-name is the same as **the P-name of ATOM**, except that all uppercase letters are converted to lowercase. If **ATOM** is not an atom, the **STRING-UPCASE** function returns **NIL**. For example:

```
    $ (STRING-DOWNCASE "Lisp is recursive")
    |lisp is recursive|
```

Here is the implementation of this function:

```
 (DEFUN STRING-DOWNCASE (ATM)
    ((ATOM ATM)
         (PACK (MAPCAR '(LAMBDA (CHAR)
                           ( (< 64 (ASCII CHAR) 91)
                             (ASCII (+ (ASCII CHAR) 32)))
                           CHAR)
                       (UNPACK ATM))) )
 )
```

9. If **ATOM** is an atom, then the **(PRINT-LENGTH ATOM)** function returns the number of characters required to print **ATOM**, based on the current value of the ***PRINT-ESCAPE*** and ***PRINT-BASE*** system variables. Otherwise, the **PRINT-LENGTH** function returns **NIL**. For example:

```
 $ (PRINT-LENGTH 'MULISP)   $ (PRINT-LENGTH NIL)
 6                          3
 $ (PRINT-LENGTH -13)
 3
```

Here is the implementation of this function:

```
 (DEFUN PRINT-LENGTH (ATM)
    ( (ATOM ATM) (LENGTH (UNPACK ATM)) )
 )
```

10. The **(FINDSTRING ATOM1 ATOM2 N)** function returns the position number of the first occurrence of **the P-name ATOM1** in **the P-name ATOM2**, with the count starting from 0. If **N** is zero or a positive integer, the search begins with **the N-th** character of **ATOM2**. If **the P-name ATOM1** is not found, the function returns **NIL**. For example:

```
 $ (FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI)
 3
 $ (FINDSTRING 'XYZ 'ABCXYZDEFXYZGHI 4)
 9
 $ (FINDSTRING 'XYZ 'ABCDEFGHI)
 NIL
```

11. If **ATOM** is either an atom or a numeric atom, **N** and **M** are non-negative integers, and **N<=M**, then the function **(SUBSTRING ATOM N M)** returns an atom whose P-name consists of the characters of the P-name **of ATOM**, starting from **the N-th** through **the M**-th, with characters counted from 0.

All values **of N** less than 0 are taken to be equal to 0.

If **M** is omitted, is a negative integer, or is greater than the number of characters in **the P-name of ATOM**, **M** is assumed to be equal to the number of characters in **the P-name**.

If **N** is greater than or equal to the number of characters in **the P-name** or if **N>M**, the **SUBSTRING** function returns the atom whose P-name is the null string.

Note that **SUBSTRING** always returns an atom, even if **ATOM** is a number. For example:

```
 $ (SUBSTRING 'ABCDEFG 2 4)     $ (SUBSTRING 'ABCDEFG 0 4)
 CDE                            ABCDE
 $ (SUBSTRING 'ABCDEFG 2)       $ (SUBSTRING 1000 0)
 CDEFG                          |1000|
```

Here is the function code:

```
  (DEFUN SUBSTRING (ATM N M)
     ( (AND (ATOM ATM) (INTEGERP N))
         ( (MINUS N)
              (SUBSTRING ATM 0 M) )
         (PACK (SUBLIST (UNPACK ATM) N M)) )
  )
```

Note. The development of the theory of data structures, methods and programming languages for list processing is mainly caused by the requirements for a particular area of computer application, namely, symbol processing. This area includes such problems as artificial intelligence, algebraic transformations, text processing and graph theory. Note that all these problems have the following common properties [1, p.444]:

- *unpredictable memory requirements. The exact total amount of memory for data for programs in this area often depends on the specific values of the data being processed, and therefore this requirement cannot be easily formulated at the time the program is written;*
- *high intensity of processing of stored data. Programs in this area usually make repeated demands to perform such operations as insertion and deletion of elements in the data structures used.*

---

(1) Tremblay J., Sorenson P. Introduction to data structures. - M.: Mashinostroenie, 1982. - 784 p.

From the next step we will start to look at ***creating and working with queues***.

Step 71. Queues

In this step we will provide ***general information about queues***.

A queue is a list structure consisting of a certain list and a Lisp cell containing pointers to the first and last elements of this list [1, p. 134].

The figures show queues consisting of the following elements:

- **A**

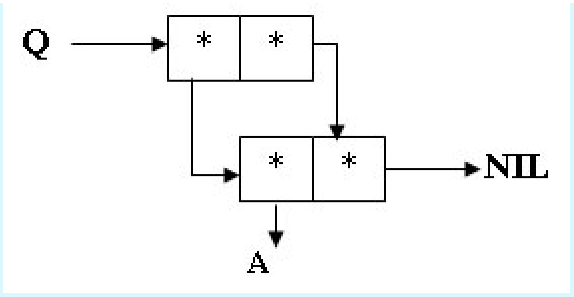


Fig. 1. First example of a queue

- **A,B**

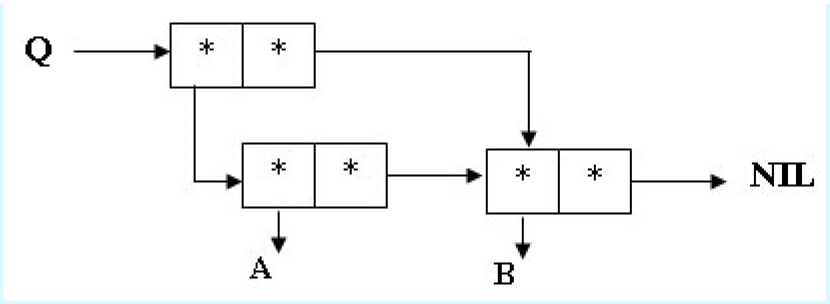


Fig.2. Second example of a queue

Example 1 [1]. The **TCONC** function places the value **X** at the end of a queue **Q**. If **Q** is an empty queue, a queue consisting of one element is formed.

```
(DEFUN TCONC (LAMBDA (X Q)
 ; Add element X to queue Q ;
   (COND ( (NULL Q) (CONS (SETQ Q (CONS X NIL)) Q) )
         (    T     (RPLACD Q (CDR (RPLACD (CDR Q)
                                           (CONS X NIL))))
         )
   )
))
```

Test examples:

```
$ (SETQ Q (TCONC A NIL))
((A) A)
$ (TCONC B Q)
((A B) B)
```

The internal representation of the value of the queue **Q** after the calculation of the first expression is shown in Fig. a), and after the second - in Fig. b).

Note that the first element of the queue **Q** is accessed by the function **(CAAR Q)**, and the last element by **(CADR Q)**.

```
$(AAR Q)
A
$ (CADR Q)
B
```

The first (but not the only) element can be removed from the queue **Q** using the call **(RPLACA Q (CDAR Q))**:

```
$ (RPLACA Q (CDAR Q))
((A) A)
```

Example 2. Let us give another (more naive) way of constructing a queue without using structure-destroying functions:

```
(DEFUN QUEUE (LAMBDA (LST)
 ; Build a queue from the list LST ;
   (LIST LST (LAST LST))
))
; --------------------- ;
(DEFUN LAST (LAMBDA (LST)
; The LAST function returns the last element of the ;
; list LST ;
   (COND ( (NULL LST) NIL )
            ( (NULL (CDR LST)) (CAR LST) )
            (        T         (LAST (CDR LST)) )
        )
    ))
```

[1] Lavrov S.S., Silagadze G.S. Automatic data processing. The Lisp language and its implementation. - M.: Nauka, 1978. - 176 p.

In the next step we will look at ***functions for building a queue***.

Step 72. Functions for building a queue

In this step we will look at ***the functions for building a queue***.

In **muLISP-83, muLISP-85**, and **muLISP-87,** there are two functions for constructing a queue: **TCONC** and **LCONC**.

The (TCONC PAIR OBJECT) function adds a **LISP OBJECT** to the end of a list whose **CAR** element points to a dotted pair **PAIR**. The list is modified. The **TCONC** function returns the dotted pair **PAIR** and a modified **CDR** element pointing to the new end of the list.

The TCONC function adds elements to the end of a list using the **RPLACD** function to modify the list. Its first argument is a dotted pair whose **CAR** element points to the beginning of the list and whose **CDR** element points to the end (the last dotted pair) of the list. The second argument is the element that was added to the end of the list. As shown in the example, if **TCONC** is called for the first time, its first argument must be **NIL** or the list **(NIL)**.

The TCONC function can be represented as:

```
  (DEFUN TCONC (PAIR OBJ)
     (SETQ OBJ (LIST OBJ))
     ( (ATOM PAIR) (CONS OBJ OBJ) )
     ( (ATOM (CDR PAIR))
           (RPLACA PAIR OBJ)
           (RPLACD PAIR OBJ) )
     ( RPLACD (CDR PAIR) OBJ )
     ( RPLACD PAIR OBJ )
  )
```

Test examples:

```
  $ (SETQ FOO NIL) $ (SETQ FOO (CONS NIL))
  NIL (NIL)
  $ (SETQ FOO (TCONC FOO 'A))   $ (TCONC FOO 'A)
  ((A) A)                       ((A) A)
  $ (TCONC FOO 'B)              $ (TCONC FOO 'B)
  ((A B) B)                     ((A B) B)
  $ (TCONC FOO 'C)              $ (TCONC FOO 'C)
  ((A B C) C)                   ((A B C) C)
  $ (CAR FOO)                   $ (CAR FOO)
  (A B C)                       (A B C)
```

The LCONC PAIR LIST function appends a **LIST** to the end of the list pointed to by **the CAR** element of a dotted pair **PAIR** by modifying the list. The **LCONC** function returns a dotted pair whose **CDR** element is modified to point to the new end of the list. The **LCONC** function appends new lists to the tail end of a list using **RPLACD** to modify the list. Its first argument is a dotted pair whose **CAR** element points to the beginning of the list and whose **CDR** element points to the end (i.e., the last dotted pair) of the list. The second argument to the **LCONC** function is the element that was appended to the end of the list.

As shown in the examples, when **LCONC** is initially called, its first argument will be a **NIL** atom or a list **(NIL)**. For example:

```
    $ (SETQ FOO NIL) $ (SETQ FOO (CONS NIL))
    NIL (NIL)
    $ (SETQ FOO (LCONC FOO '(A B)))  $ (LCONC FOO '(A B))
    ((A B) B)                        ((A B) B)
    $ (LCONC FOO '(C D))             $ (LCONC FOO '(C D))
    ((A B C D) D)                    ((A B C D) D)
    $ (LCONC FOO '(E F))             $ (LCONC FOO '(E F))
    ((A B C D E F) F)                ((A B C D E F) F)
    $ (CAR FOO)                      $ (CAR FOO)
    (A B C D E F )                   (A B C D E F)
```

The implementation of the function looks like this:

```
   (DEFUN LCONC (PAIR LST)
      ( (ATOM LST) PAIR )
      ( (ATOM PAIR) (CONS LST (LAST LST)) )
      ( (ATOM (CDR PAIR)) (RPLACA PAIR LST)
                          (RPLACD PAIR (LAST LST)) )
      ( RPLACD (CDR PAIR) LST )
      ( RPLACD PAIR (LAST LST) )
   )
```

In the next step we will look at ***cyclic (ring) lists***.

Step 73.
Cyclic (ring) lists

In this step we will look at ***constructing circular lists***.

All the list structures we have looked at so far have been cycle-free.

A ***cyclic (ring) list*** is a list in which the pointer from a certain cell points to a place in the list from which the given cell can be reached again [2, p.37]. Cyclic lists do not allow direct representation in list notation, but can be depicted using graphical notation.

For example, the following lists are cyclic:

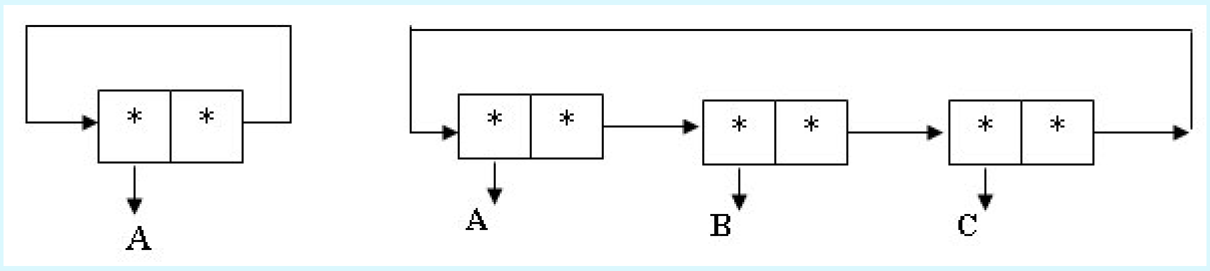


Fig. 1. Examples of cyclic lists

Such lists cannot be constructed if the only way to form new initial cells is the **CONS** procedure. Cyclicity can only be introduced using the "Replace functions" **RPLACA** and **RPLACD**, which change the existing list "physically".

Example 1. Building a circular list.

Using the **RPLACD** function, you can define a **CIRCLE** function that turns an list **LST** into a circular list.

```
(DEFUN CIRCLE (LAMBDA (LST)
   (MAKE_CIRCLE LST LST)
))
; ------------------------------ ;
(DEFUN MAKE_CIRCLE (LAMBDA (LST Y)
   ( (NULL LST) LST)
   ( (NULL (CDR LST)) (RPLACD LST Y) )
   ( MAKE_CIRCLE (CDR LST) Y )
))
```

Example 2 [1, p.136].

The LCYCLEP predicate returns the value **T** if the argument value contains a cycle along the chain of **CDD** pointers, and **NIL** otherwise (repeatedly applying the **CDR** function to the argument value can result in an atom).

```
( DEFUN LCYCLE (LAMBDA ( X )
   (AND (NOT (ATOM X)) (NOT (ATOM (CDR X)))
        (LCYCLE1 (CDR X) (CDDR X)))
))
; ------------------------ ;
(DEFUN LCYCLE1 (LAMBDA (X Y)
   (OR (EQ X Y) (AND (NOT (ATOM Y))
                     (NOT (ATOM (CDR Y)))
                     (LCYCLE1 (CDR X) (CDDR Y)))
   )
))
```

Test examples:

```
$ (LCYCLEP (A B C))
NIL
$ (LCYCLEP (RPLACD (CDDR (A B C)) (A B C)))
T
$ (LCYCLEP (CONS X NIL))
NIL
```

Example 3 [1, p.137].

The CYCLEP predicate returns the value **T** if the argument value is a cyclic list structure, and the value **NIL** if there are no cycles (cyclic pointer chains) in the argument value. It uses the **CYCLE1** function, the first argument of which is one of the nodes (substructures) of the list structure examined in **CYCLEP**, the second argument is the path from this node to the original node, and the third is the list of nodes already examined.

```
( DEFUN CYCLE (LAMBDA ( X )
   (NULL (CYCLE1 X NIL T))
))
; ------------------------- ;
(DEFUN CYCLE1 (LAMBDA (X U V)
   (COND ( (ATOM X) V )
         ( (MEMBER X U) NIL )
         ( (MEMBER X V) V )
         ( (NULL (SETQ V (CYCLE1 (CAR X)
                                 (SETQ U (CONS X U))
                                  V))
           )
                 NIL )
         ( (NULL (SETQ V (CYCLE1 (CDR X) U V))) NIL )
         (  T  (CONS X V) )
   )
))
```

---

[1] Lavrov S.S., Silagadze G.S. Automatic data processing. The Lisp language and its implementation. - M.: Nauka, 1978. - 176 p.
[2] Foster J. List processing. - M.: Mir, 1974. - 72 p.

In the next step we will get acquainted ***with sets***.

Step 74.
Sets

In this step we will look at ***working with sets***.

The subject area, the problems of which are naturally formulated in terms of working with sets, turns out to be quite broad. This is not surprising, since ***the language of set theory is the universal language of mathematics***.

The data type "set" is not typical for programming languages. This concept is more familiar to mathematicians than to programmers. However, it turns out that for a number of applications sets are natural objects. Of course, it should be understood that sets are an ideal object, and virtually any computer works with a "set" as with an ordered set of data. However, this is not yet a reason to force the programmer to work with ordered objects: let the language give him the opportunity to operate with sets.

In this step, lists are treated as sets of their elements - the order of elements in the list is not important, and two or more identical list elements are treated as one element of the set.

Example 1. Library of functions for working with sets.

```
   (DEFUN LIST-SET (LAMBDA (LST)
    ; The LIST-SET function converts a list of LSTs into a set ;
      (COND ( (NULL LST) NIL )
            ( (MEMBER (CAR LST) (CDR LST))
                (LIST-SET (CDR LST)) )
            (  T  (CONS (CAR LST) (LIST-SET (CDR LST))) )
      )
   ))
   ; --------------------- ;
   (DEFUN SETP (LAMBDA (LST)
   ; The SETP predicate checks whether the list LST is a ;
   ; set, i.e. whether each element of the LST appears in the ;
   ; list only once ;
      (COND ( (NULL LST)                     T  )
            ( (MEMBER (CAR LST) (CDR LST)) NIL )
            (  T  (SETP (CDR LST)) )
      )
   ))
   ; ---------------------------- ;
```

```
(DEFUN UNION (LAMBDA (LST1 LST2)
;The UNION function returns the union of ;
;sets LST1 and LST2 ;
   (COND ( (NULL LST1) LST2 )
         ( (MEMBER (CAR LST1) LST2)
             (UNION (CDR LST1) LST2) )
         (  T  (CONS (CAR LST1) (UNION (CDR LST1) LST2)) )
   )
))
; ------------------------------------ ;
(DEFUN INTERSECTION (LAMBDA (LST1 LST2)
; The INTERSECTION function returns the intersection of ;
; the sets LST1 and LST2 ;
   (COND ( (NULL LST1) NIL )
         ( (MEMBER (CAR LST1) LST2)
             (CONS (CAR LST1)
                   (INTERSECTION (CDR LST1) LST2)) )
         (  T  (INTERSECTION (CDR LST1) LST2) )
   )
))
; ------------------------------ ;
(DEFUN DIFFER (LAMBDA (LST1 LST2)
; The DIFFER function returns the difference between the sets LST1 and ;
; LST2, i.e. it removes from the set LST1 all elements common with
the set LST2 ;
   (COND ( (NULL LST1) NIL  )
         ( (NULL LST2) LST1 )
         ( (MEMBER (CAR LST1) LST2)
             (DIFFER (CDR LST1) LST2) )
         (  T  (CONS (CAR LST1)
                     (DIFFER (CDR LST1) LST2)) )
   )
))
; ---------------------------------- ;
(DEFUN SYMDIFFER1 (LAMBDA (LST1 LST2)
; The SYMDIFFER1 function returns the symmetric
difference of ; ; sets LST1 and LST2 (the set
consisting of all elements of ; ; sets LST1 and LST2
that belong to either LST1 or LST2, but not to the ;
; intersection of LST1 and LST2 ;
   (COND ( (NULL LST1) LST2 )
         ( (MEMBER (CAR LST1) LST2)
             (SYMDIFFER1 (CDR LST1)
                         (REMOVE (CAR LST1) LST2)) )
         (  T  (CONS (CAR LST1)
                     (SYMDIFFER1 (CDR LST1) LST2)) )
   )
))
; ---------------------------------- ;
(DEFUN SYMDIFFER2 (LAMBDA (LST1 LST2)
; The SYMDIFFER2 function returns the symmetric
difference of ; ; sets LST1 and LST2 (the set
consisting of all elements of ; ; sets LST1 and LST2
that belong to either LST1 or LST2, but not to the ;
; intersection of LST1 and LST2 ;
   (COND ( (NULL LST1) LST2 )
         (  T  (DIFFER (UNION LST1 LST2)
                       (INTERSECTION LST1 LST2)) )
   )
))
; --------------------------------- ;
(DEFUN EQUALSET1 (LAMBDA (LST1 LST2)
; The predicate EQUALSET1 tests the equality of two ;
; sets LST1 and LST2 ;
   (COND ( (NULL LST1) (NULL LST2) )
         ( (MEMBER (CAR LST1) LST2)
                 (EQUALSET1 (CDR LST1)
                            (REMOVE (CAR LST1) LST2)) )
         ( T  NIL )
   )
))
; -------------------------------- ;
(DEFUN EQUALSET2 (LAMBDA (LST1 LST2)
; The predicate EQUALSET2 tests the equality of two ;
; sets LST1 and LST2 ;
   (AND (SUBSET LST1 LST2) (SUBSET LST2 LST1))
))
; ----------------------------- ;
(DEFUN SUBSET (LAMBDA (LST1 LST2)
; The SUBSET predicate tests whether the set LST1 ;
; is a subset of the set LST2, in other words, it ;
; returns a value T if every element of the list LST1 ;
; is contained in the list LST2 ;
   (COND ( (NULL LST2) (NULL LST1) )
         ( (NULL LST1) T )
         ( (MEMBER (CAR LST1) LST2)
              (SUBSET (CDR LST1) LST2) )
         ( T  NIL )
   )
))
; ----------------------------------- ;
(DEFUN NONINTERSECT (LAMBDA (LST1 LST2)
; The NONINTERSECT predicate checks that the sets ;
; LST1 and LST2 are disjoint ;
   (COND ( (NULL LST1) T )
         ( (MEMBER (CAR LST1) LST2) NIL )
         (  T  (NONINTERSECT (CDR LST1) LST2) )
   )
))
; --------------------------- ;
(DEFUN REMOVE (LAMBDA (ATM LST)
; The REMOVE function returns a list in which
all occurrences of the ATM element in the list LST have been removed ; ;
   (COND ( (NULL LST) NIL )
         ( (EQ ATM (CAR LST)) (REMOVE ATM (CDR LST)) )
         (  T  (CONS (CAR LST) (REMOVE ATM (CDR LST))) )
   )
))
```

Example 2. The function returns a list of common characters (not just letters!) of two strings.

```
(DEFUN COMMON-LETTER (LAMBDA (W1 W2)
 ; W1 and W2 are literal atoms (words) ;
   (LIST-SET (INTERSECTION (UNPACK W1) (UNPACK W2)))
))
; ------------------------- ;
(DEFUN LIST-SET (LAMBDA (LST)
; The LIST-SET function converts an list LST into a set ;
   (COND ( (NULL LST) NIL )
         ( (MEMBER (CAR LST) (CDR LST))
             (LIST-SET (CDR LST)) )
         (  T  (CONS (CAR LST) (LIST-SET (CDR LST))) )
   )
))
; ----------------------------------- ;
(DEFUN INTERSECTION (LAMBDA (LST1 LST2)
; The INTERSECTION function returns the intersection of ;
; the sets LST1 and LST2 ;
   (COND ( (NULL LST1) NIL )
         ( (MEMBER (CAR LST1) LST2)
                (CONS (CAR LST1)
                      (INTERSECTION (CDR LST1) LST2)) )
            (  T  (INTERSECTION (CDR LST1) LST2) )
      )
   ))
```

Example 3. Consider the problem of ***generating all possible permutations of a set of* N *elements***. (It is assumed that the set is given as a list of pairwise distinct elements.)

Let there be a program that can generate a list of all permutations of a set of **(N-1)** elements. Then, to construct all permutations of a set of **N** elements, one must successively select one of the elements of this set and add it to the first place in each of the permutations of the set of **(N-1)** elements obtained from the original set by deleting the selected element. By our assumption, a program that generates all permutations of a set of **(N-1)** elements already exists. To this we must add a program that generates all permutations of the empty set. This is easy to do. The result of this program should be a list consisting of one element - the empty set.

The rule for generating permutations can be written as follows.

Let **M** be a set consisting of pairwise distinct elements, all permutations of which need to be constructed. Then [1, p.56-57]:

```
   < All permutations of the set M > ::=
    if M is the empty set, then (NIL),
    otherwise
       for all X belonging to M,
         Combine
            (Add X to the first place in each of
               < All permutations of the set M without the element X >)
```

Let us list the functions required to solve the problem:

```
   (DEFUN PERMLIST (LAMBDA (M)
    ; Generate all permutations of elements of list M ;
      (COND ( (NULL M) (LIST NIL) )
            (  T  (PERM2 M M) )
      )
   ))
   ; --------------------------- ;
   (DEFUN PERM2 (LAMBDA (UNUSED M)
   ; "For all X belonging to M, combine..." ;
      (COND ( (NULL UNUSED) NIL )
            (  T  (APPEND (MULTICONS
                             (CAR UNUSED)
                             (PERMLIST (DELETE (CAR UNUSED) M)))
                          (PERM2 (CDR UNUSED) M)
                )
            )
      )
   ))
   ; ------------------------------ ;
   (DEFUN MULTICONS (LAMBDA (X LISTS)
   ; Append element X to each list ;
   ; from list LISTS ;
      (COND ( (NULL LISTS) NIL )
            (  T  (CONS (CONS X (CAR LISTS))
                        (MULTICONS X (CDR LISTS))) )
      )
   ))
   ; ----------------------- ;
   (DEFUN APPEND (LAMBDA (U V)
   ; Merge two lists U and V ;
```

```
      (COND ( (NULL U) V )
            (  T  (CONS (CAR U) (APPEND (CDR U) V)) )
      )
   ))
   ; ----------------------- ;
   (DEFUN DELETE (LAMBDA (X U)
   ; Remove the first occurrence of element X from list U ;
   ; at the top level ;
      (COND ( (NULL U) NIL )
            ( (EQ (CAR U) X) (CDR U) )
            (  T  (CONS (CAR U) (DELETE X (CDR U))) )
      )
   ))
```

[1] Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.

In the next step we will start to get acquainted with ***binary search trees***.

Step 75.
Binary Search Trees

In this step we will consider ***algorithms for creating binary search trees and provide a library of functions for working with binary trees***.

To represent binary trees, we will use two methods, each of which is based on a specific list representation of the tree.

The first way.

A binary search tree consists of nodes of the form:

```
        (Root (Left-subtree Right-subtree))
```

At each node, the following condition is satisfied: all elements from the nodes of its left subtree in some order (for example, by numerical value or in alphabetical order) precede the element from the node and, accordingly, elements from the nodes of the right subtree follow them.

Example 1.

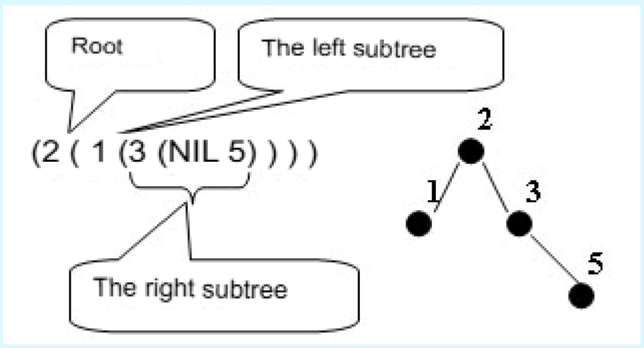


Fig.1. Example 1

Example 2.

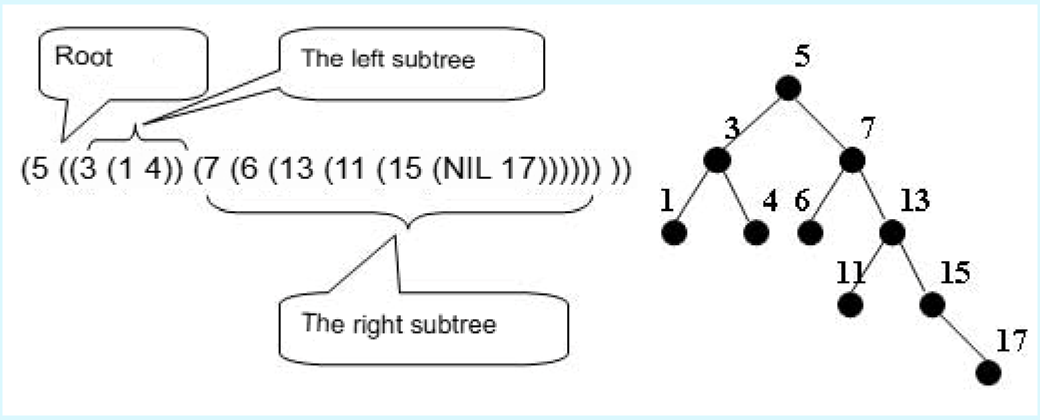


Fig.2. Example 2

Note that if **TREE** has a representation of the form

```
        (Root (Left-subtree Right-subtree))
```

that

```
      (Root    (Left-subtree     Right-subtree))
      ------   ---------------   ----------------
         ^          ^                  ^
         ¦          ¦                  ¦
    (CAR TREE)  (CAR (CADR TREE))  (CADR (CADR TREE))
```

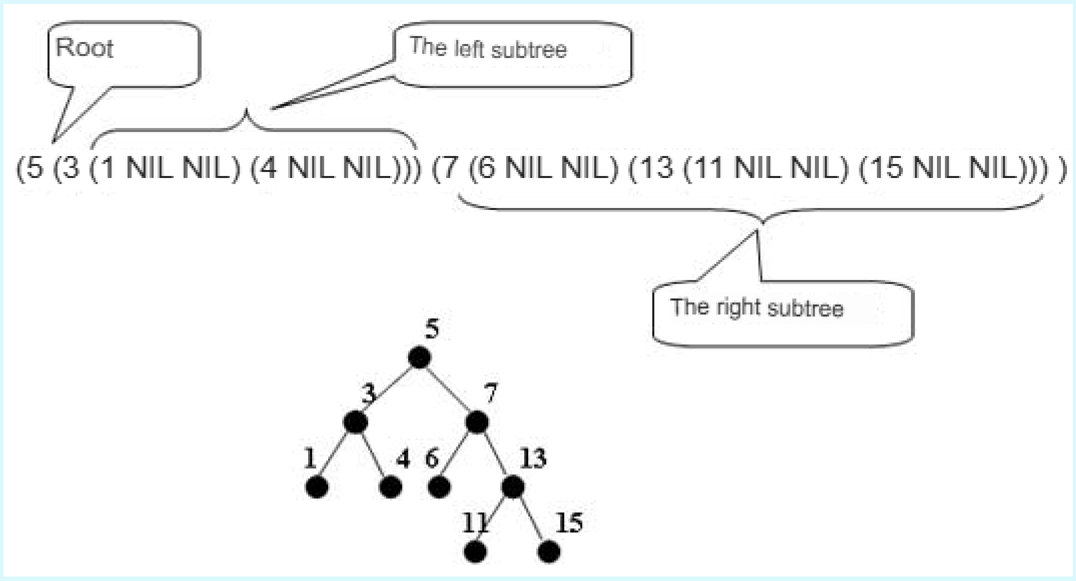
The second way [1].

A binary search tree consists of nodes of the form:

```
         (Element Left-subtree Right-subtree)
```

At each node, the following condition is satisfied: all elements from the nodes of its left subtree in some order (for example, by numerical value or in alphabetical order) precede the element from the node and, accordingly, elements from the nodes of the right subtree follow them.

Example 3.


Root
The left subtree
(5 (3 (1 NIL NIL) (4 NIL NIL))) (7 (6 NIL NIL) (13 (11 NIL NIL) (15 NIL NIL))))
The right subtree
5
3
7
1
4
6
13
11
15
Fig.3. Example 3

Note that if **TREE** has a representation of the form

```
         (Root Left-subtree Right-subtree)
```

that

```
        ( Root    Left-subtree     Right-subtree )
          ------  ---------------  ----------------
            ^           ^                 ^
            ¦           ¦                 ¦
        (CAR TREE)   (CADR TREE)      (CADDR TREE)
```

Example 4. Library for working with binary trees.

```
   (DEFUN TEST (LAMBDA NIL
      (PRINT "Let's build a tree with a counter of repeated elements!")
      (SETQ TREE NIL)
      (LOOP
         (PRINT "Enter the next tree element:")
         (SETQ A (READ)) ( (EQ A '!) )
         (PRINT (SETQ TREE (ADDTREE1 A TREE)))
         (PRINT "----------------------------")
      )
      (PRINT "-----------------------------------")
      (PRINT "Building tree: ") (SETQ TREE NIL)
      (LOOP
         (PRINT "Enter the next tree element:")
         (SETQ A (READ)) ( (EQ A '!) )
         (PRINT (SETQ TREE (ADDTREE A TREE)))
      )
      (PRINT "------------------------------")
      (PRIN1 "Tree root:                    ")
         (PRINT (ROOT TREE))
      (PRIN1 "Left subtree:                 ")
         (PRINT (LEFT TREE))
      (PRIN1 "Right subtree:                ")
         (PRINT (RIGHT TREE))
      (PRIN1 "Breadth-first tree traversal:")
         (PRINT (REMBER
                   NIL
                   (LISTATOMS (UNTREE (TOP TREE) TREE))))
      (PRIN1 "Left-hand tree traversal:     ")
         (PRINT (UNTREE1 TREE))
      (PRIN1 "Number of levels in tree:     ")
         (PRINT (TOP TREE))
      (PRIN1 "Number of leaves in a tree:   ")
         (PRINT (NLIST TREE))
      (PRIN1 "Tree copy: ")
         (PRINT (TCOPY TREE))
      (PRINT "------------------------------")
      (PRINT "Let's start searching for an element in the tree!")
      (LOOP
         (PRINT "Enter the tree element you are looking for:")
         (SETQ A (READ))
         ( (EQ A '!) )
         (PRINT (SEARCH A TREE))
      )
      (PRINT "-------------------------------")
      (PRINT "Let's proceed to deleting the element!")
      (LOOP
         (PRINT "Enter tree item to remove: ")
         (SETQ A (READ))
         ( (EQ A '!) )
         (PRINT (SETQ TREE (DELETE A TREE)))
      )
      (PRINT "-------------------------------")
      (PRINT "Let's proceed to deleting the element in a different way!")
      (LOOP
         (PRINT "Enter tree item to remove:")
         (SETQ A (READ))
         ( (EQ A '!) )
         (PRINT (SETQ TREE (DELETE1 A TREE)))
      )
      (PRINT "-----------------------------------")
      (PRINT "Let's start selecting subtrees!")
      (LOOP
         (PRINT "Enter any tree element:")
         (SETQ A (READ))
         ( (EQ A '!) 'END )
         (PRINT (PRETREE A TREE))
         (PRINT (POSTTREE A TREE))
         (PRINT (UNITETREE (PRETREE A TREE)
                           (POSTTREE A TREE)))
         (PRINT "----------------------------")
      )
   ))
   ; ------------------------------
   (DEFUN ADDTREE1 (LAMBDA (A TREE)
   ; The ADDTREE function adds an element A to the search tree TREE
   ; counting the number of times the element A is repeated during input
      (COND ( (NULL TREE) (LIST (CONS A 0) NIL NIL) )
            ( (EQUAL A (CAAR TREE))
                    (LIST (CONS A (+ (CDAR TREE) 1))
                          (CADR TREE) (CADDR TREE)) )
            ( (< A (AAR TREE))
                    (LIST (CAR TREE) (ADDTREE1 A (CADR TREE))
                          (CADDR TREE)) )
            (   T   (LIST (CAR TREE)
                          (CADR TREE) (ADDTREE1 A (CADDR TREE))) )
      )
   ))
   ; -----------------------------
   (DEFUN ADDTREE (LAMBDA (A TREE)
   ; The ADDTREE function adds an element A to the search tree TREE
      (COND ( (NULL TREE) (LIST A NIL NIL) )
            ( (EQUAL A (CAR TREE)) TREE )
            ( (< A (CAR TREE))
                    (LIST (CAR TREE) (ADDTREE A (CADR TREE))
                          (CADDR TREE)) )
            (   T   (LIST (CAR TREE) (CADR TREE)
                          (ADDTREE A (CADDR TREE))) )
      )
   ))
   ; -------------------------
   (DEFUN CONSTR (LAMBDA (LST)
   ; The CONSTR function builds a tree from the list LST in reverse order
      (COND ( (NULL LST) NIL )
            (  T  (ADDTREE (CAR LST) (CONSTR (CDR LST))) )
      )
   ))
   ; ---------------------------
   (DEFUN CONSTREE (LAMBDA (LST)
   ; The CONSTR function builds a tree from the list LST in preorder
      (CONSTR (REVERSE LST))
   ))
   ; ----------------------------
   (DEFUN SEARCH (LAMBDA (A TREE)
   ; The SEARCH function searches for element A in the TREE
 . ; If successful, the function returns the subtree of
   the TREE in which element A is the root; if
   the search fails, the function returns NIL.
      (COND ( (NULL TREE)          NIL  )
            ( (EQUAL A (CAR TREE)) TREE )
         ( (< A (CAR TREE)) (SEARCH A (CADR  TREE)) )
         (          T           (SEARCH A (CADDR TREE)) )
   )
))
; ----------------------------------
(DEFUN REPLACE (LAMBDA (OLD NEW LST)
; Replacement of the OLD sublist with the NEW sublist in the list LST
   (COND ( (ATOM LST) LST )
         ( (EQUAL OLD LST) NEW )
         (  T  (CONS (REPLACE OLD NEW (CAR LST))
                     (REPLACE OLD NEW (CDR LST))) )
   )
))
; ------------------------
(DEFUN ROOT (LAMBDA (TREE)
; The ROOT function returns the root of the TREE tree
   (CAR TREE)
))
; ------------------------
(DEFUN LEFT (LAMBDA (TREE)
; The function returns the left subtree of the TREE tree
   (CADR TREE)
))
; -------------------------
(DEFUN RIGHT (LAMBDA (TREE)
; The function returns the right subtree of the TREE tree
   (CADDR TREE)
))
; -----------------------------
(DEFUN RIGHTLIST (LAMBDA (TREE)
; Returns the rightmost leaf of the TREE
   (COND ( (NULL (RIGHT TREE)) (CAR TREE) )
         (  T  (RIGHTLIST (RIGHT TREE)) )
   )
))
; ----------------------------
(DEFUN LEFTLIST (LAMBDA (TREE)
; Returns the leftmost leaf of the TREE
    (COND ( (NULL (LEFT TREE)) (CAR TREE) )
          (  T  (LEFTLIST (LEFT TREE)) )
    )
))
; ------------------------------
(DEFUN DELETE (LAMBDA (ATM TREE)
; Removing an ATM node from a TREE
;  (non-recursive delete option)
   (SETQ SUBTREE (SEARCH ATM TREE))
   (COND ( (NULL SUBTREE) (PRINT "No node in tree!") )
         (  T
              ; Node ATM in tree TREE found
              (COND ( (EQUAL SUBTREE (LIST ATM NIL NIL))
                       ; The node found is a leaf
                       (REPLACE SUBTREE NIL TREE)
                    )
                    ( (AND (NOT (NULL (LEFT  SUBTREE)))
                           (NOT (NULL (RIGHT SUBTREE))))
                      ; The found node has both subtrees
                         (SETQ NODE
                           (RIGHTLIST (LEFT SUBTREE)))
                         (RPLACA SUBTREE UZEL)
                         (COND ( (NULL (RIGHT
                                        (LEFT SUBTREE)))
                                 (REPLACE (LEFT SUBTREE)
                                   (CADR (LEFT SUBTREE))
                                          TREE) )
```

```
                                ( (NULL (LEFT
                                         (LEFT SUBTREE)))
                                  (REPLACE (LEFT SUBTREE)
                                    (CADDR (LEFT SUBTREE))
                                           TREE) )
                                ( T (REPLACE (SHEET NODE
                                                NIL NIL)
                                               NIL TREE) )
                          )
                    )
                    ( (NULL (RIGHT SUBTREE))
                      ; The found node has only a left subtree
                          (REPLACE SUBTREE (CADR SUBTREE)
                                   TREE)
                    )
                    ( (NULL (LEFT SUBTREE))
                      ; The found node has only the right subtree
                          (REPLACE SUBTREE (CADDR SUBTREE)
                                   TREE)
                    )
              )
         )
   )
))
; -------------------------------
(DEFUN DELETE1 (LAMBDA (ATM TREE)
; Removing an ATM node from a TREE
; (recursive deletion option)
   (COND ( (NULL TREE) NIL )
         ( (< ATM (ROOT TREE))
              (LIST (CAR TREE)
                    (DELETE1 ATM (LEFT TREE))
                    (RIGHT TREE))
         )
         ( (> ATM (ROOT TREE))
              (LIST (CAR TREE)
                    (LEFT TREE)
                    (DELETE1 ATM (RIGHT TREE)))
         )
         (  T  (COND ( (NULL (RIGHT TREE)) (LEFT  TREE) )
                     ( (NULL (LEFT  TREE)) (RIGHT TREE) )
                     (  T  (LIST (UD (LEFT TREE))
                                 (DELETE1
                                     (UD (LEFT TREE))
                                     (LEFT TREE))
                                 (RIGHT TREE)) )) )
   )
))
; ----------------------
(DEFUN UD (LAMBDA (TREE)
; Helper function for the DELETE1 function
   (COND ( (NULL (RIGHT TREE)) (CAR TREE) )
         (  T  (UD (RIGHT TREE)) )
   )
))
; -----------------------
(DEFUN TOP (LAMBDA (TREE)
; The TOP function returns the number of levels in the tree TREE
; (the root of the tree is at level zero)
   (COND ( (NULL TREE) -1 )
         (  T  (+ 1 (MAX (TOP (LEFT  TREE))
                         (TOP (RIGHT TREE)))) )
   )
))
; ----------------------
```

```
   (DEFUN MAX (LAMBDA (M N)
   ; The MAX function returns the larger of the numbers M and N
      (COND ( (> M N) M )
            ( T  N )
      )
   ))
   ; -------------------------
   (DEFUN NLIST (LAMBDA (TREE)
   ; The NLIST function returns the number of leaves in a TREE
      (COND ( (NULL TREE) 0 )
            ( (EQUAL (CDR TREE) (LIST NIL NIL)) 1 )
            (  T  (+ (NLIST (LEFT TREE)) (NLIST (RIGHT TREE))) )
      )
   ))
   ; -------------------------
   (DEFUN TCOPY (LAMBDA (TREE)
   ; The TCOPY function returns a copy of the TREE
      (COND ( (ATOM TREE) TREE )
            (  T  (CONS (TCOPY (CAR TREE))
                        (TCOPY (CDR TREE))) )
      )
   ))
   ; -----------------------------
   (DEFUN PRETREE (LAMBDA (A TREE)

   ; The PRETREE function extracts all nodes preceding a given element A from the tree
; TREE into a separate tree
      (COND ( (NULL TREE) NIL )
            ( (< (CAR TREE) A)
                    (LIST (CAR TREE) (CADR TREE)
                          (PRETREE A (CADDR TREE))) )
            (   T   (PRETREE A (CADR TREE)) )
      )
   ))
   ; ------------------------------
   (DEFUN POSTTREE (LAMBDA (A TREE)
   ; The POSTTREE function extracts into a separate tree from the tree
   ; TREE all nodes following a given element A
      (COND ( (NULL TREE) NIL )
            ( (< (CAR TREE) A)
                    (POSTTREE A (CADDR TREE)) )
            (   T   (LIST (CAR TREE)
                          (POSTTREE A (CADR TREE))
                          (CADDR TREE)) )
      )
   ))
   ; ------------------------------------
   (DEFUN UNITETREE (LAMBDA (TREE1 TREE2)
   ; The UNITETREE function combines two search trees
   ; TREE1 and TREE2 into one search tree
      (COND ( (NULL TREE1) TREE2 )
            ( (NULL TREE2) TREE1 )
            (   T   (LIST (CAR TREE1)
                          (UNITREE (PRETREE (CAR TREE1)
                                                TREE2)
                                   (CADR  TREE1))
                          (UNITETREE (POSTTREE (CAR TREE1)
                                                TREE2)
                                   (CADDR TREE1))) ) )
   ))
   ; ----------------------------
   (DEFUN UNTREE (LAMBDA (M TREE)
   ; "Dirty" breadth-first traversal of the TREE tree, starting
   ; from level 0 to level M
      (COND ( (EQ M 0) (CAR TREE) )
            (  T  (LIST (UNTREE (- M 1) TREE)
                        (SEE M TREE)) )
      )
   ))
   ; ----------------------------
   (DEFUN UNTREE1 (LAMBDA (TREE)
   ; Left-hand traversal of a tree TREE
       (REMBER NIL (LISTATOMS TREE))
   ))
   ; --------------------------
   (DEFUN SEE (LAMBDA (N TREE)
   ; Traverses a TREE in breadth and creates a "dirty"
   ; list containing the N-th level nodes of the tree
      (COND ( (EQ N 0) (CAR TREE) )
            ( (EQ N 1)
                 (LIST (CAR (CADR TREE)) (CAR (CADDR TREE)))
            )
            (  T  (LIST (SEE (- N 1) (CADR  TREE))
                        (SEE (- N 1) (CADDR TREE))) )
      )
   ))
   ; ------------------------------
   (DEFUN LISTATOMS (LAMBDA (TREE)
   ; The LISTATOMS function returns a list of
   ; elements (including NIL !) contained in the search tree TREE
      (COND ( (NULL TREE) NIL)
            ( (ATOM (CAR TREE))
                 (CONS (CAR TREE) (LISTATOMS (CDR TREE))) )
            (  T  (APPEND (LISTATOMS (CAR TREE))
                          (LISTATOMS (CDR TREE))) )
      )
   ))
   ; ------------------------------
   (DEFUN REMBER (LAMBDA (ATM LST)
   ; The REMBER function returns a list with
   all occurrences of the ATM element in the list LST removed
      (COND ( (NULL LST) NIL )
            ( (EQ ATM (CAR LST)) (REMBER ATM (CDR LST)) )
            (  T  (CONS (CAR LST) (REMBER ATM (CDR LST))) )
      )
   ))
   ; --------------------------------
   (DEFUN APPEND (LAMBDA (LST1 LST2)
   ; The APPEND function returns a list consisting of
   ; the elements of LST1 appended to LST2
      (COND ( (NULL LST1) LST2 )
            ( (NULL LST2) LST1 )
            (  T  (CONS (CAR LST1)
                        (APPEND (CDR LST1) LST2)) )
      )
   ))
```

*Note: The **DELETE1** and **UD** functions are "literally" rewritten from the following recursive procedure written **by N. Wirth** [2]:*

```
  PROCEDURE   U_d_a_l_d_r ( var d: Ref; k: Integer );
   {Remove node with key k from tree d. }
  { Ref - type pointer to tree node }
     var   q: Ref;
     { -------------------------- }
     PROCEDURE   U_d ( var r: Ref);
      BEGIN
        If   r^.Right= Nil
           then   begin
                    q^.Key := r^.Key; q^.Count := r^.Count;
                    q := r;
                    r := r^.Left; { Discarded the node... }
                    Dispose (q)     { Freed the memory }
                 end
          else   U_d (r^.Right)
      END ;
   { ----- }
 BEGIN
   If   d= Nil
    then   { First case of the delete algorithm }
        Writeln ('Node with the given key was not found in the tree...')
    else  { 1Search for a node with the given key 0 }
      If   k<d^.Key
          then   U_d_a_l_d_r (d^.Left,k)
          else   If   k>d^.Key
                 then   U_d_a_l_d_r (d^.Right,k)
                 else
                  begin  { Node found, must be deleted }
                        { Second case of the deletion algorithm }
                        q := d;
                        If   q^.Right = Nil
                           then   d := q^.Left
                            else   If   q^.Left= Nil
                                  then   d := q^.Right
                                   else   { Tpetiy case }
                                        { removal algorithm }
                                        U_d (q^.Left)
                  end
END ;
```

[1] Hyvänen E., Seppänen J. The World of Lisp. In 2 volumes. Volume 1: Introduction to the Lisp language and functional programming. - Moscow: Mir, 1990. - 447 p.
[2] Wirth N. Algorithms + data structures = programs. - Moscow: Mir, 1985. - 406 p.

In the next step we will get acquainted with **TREE** ***structures***.

Step 76.
TRIE structures

In this step we will consider what **TRIE structures** ***are and what their area of application is***.

TRIE ***structures (dictionaries, borts)*** are a class of data structures in which the value of the key by which the search is performed is considered as an ordered sequence (string) of characters from a certain alphabet containing **V** symbols [1, pp. 134-142; 2, pp. 572-600]. The name **"TRIE"** itself comes from the word **"reTRIEval"** ***(search, selection)***.

Consider, for example, the "letter-by-letter cutouts" found in many dictionaries; given the first letter of a given word, we can immediately find the pages containing all the words beginning with that letter.

A TRIE structure is ***a highly branching tree*** in which exactly **V** branches emerge from each node, one for each possible character in the key string position. If the key consists of **K** characters (bytes), then searching for the corresponding data record to determine whether it is present or absent from **the TRIE** structure requires scanning **the K** nodes of the tree. The data records in this structure correspond to leaves.

The advantages of **the TRIE** structure are:

- the ability to quickly access data at the cost of additional memory costs;
- ***If a search for an entry in a tree fails, the element that best matches*** the search argument will still be found. This property is useful for some applications.

The i-th position in the key can have **V** possible values, each value corresponding to a specific position in a specific node at the **i-th** level of the tree structure. This addressing method is a type of ***direct addressing***, in which the value of a symbol implicitly determines the position in the node where the value of the corresponding pointer can be found. The value of the pointer determines the location of the node at the next level, which represents the possible values of the (**i + 1**)-th symbol in the key. Therefore, for a key of **K characters, only K** nodes are scanned, one at each of **the K** levels of the tree.

Example 1. Let the following words be entered into the dictionary sequentially: ***Mas, Mal, Myach***.

The arrangement of information in the dictionary can be represented as follows:

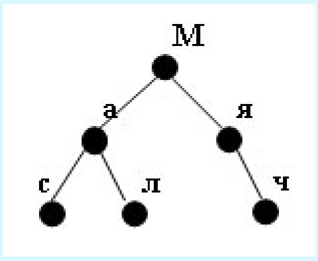


Fig. 1. Location of information in the dictionary

The location of information in a Lisp list:

```
(M (a (s l) I (h)))
```

Example 2. Let the following words be entered into **the TRIE structure sequentially:** ***Natasha, Natashenka, Natalich, Natulya, Natusya, Tatosha, Tatoshka, Tuska***.

The arrangement of information in the dictionary can be represented as follows:

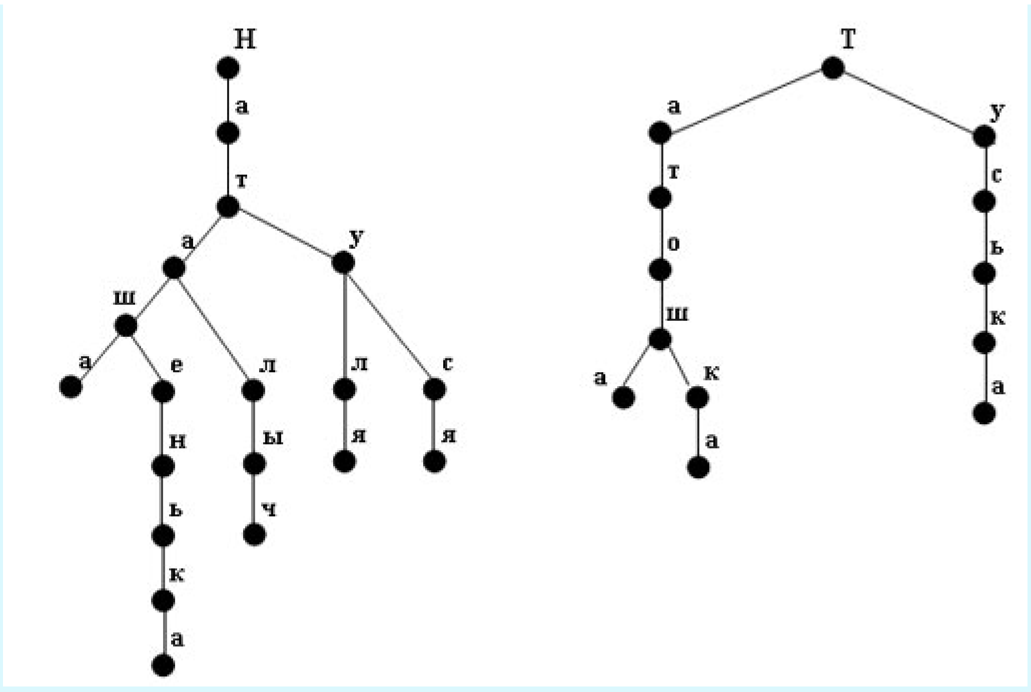


Fig.2. Location of information in the dictionary

The location of information in a Lisp list:

```
  (H (a (t (a (sh (a e (n (ь (k (a))))) l (y (ch))) u (l (ya) s (ya)))))
   T (a (t (o (w (a k (a))))) y (s (ь (k (a))))))
```

Because keys can be of variable length, **the TRIE** structure is generally unbalanced, meaning that if $\mathbf{K}_j$ denotes the length of key **j**, then for different **j** the leaves in **the TRIE** structure are located at different $\mathbf{K}_j$ levels.

Of course, each leaf of the **TRIE** structure must have ***a flag*** (with values, for example, 0 and 1), which is intended to identify the last vertex on the path corresponding to the key value.

Let us schematically depict a **TRIE** structure containing keys **AAA, AAC, AC, BAB, BBCA, CA, CB**, composed of symbols of the alphabet **{A, B, C}**:

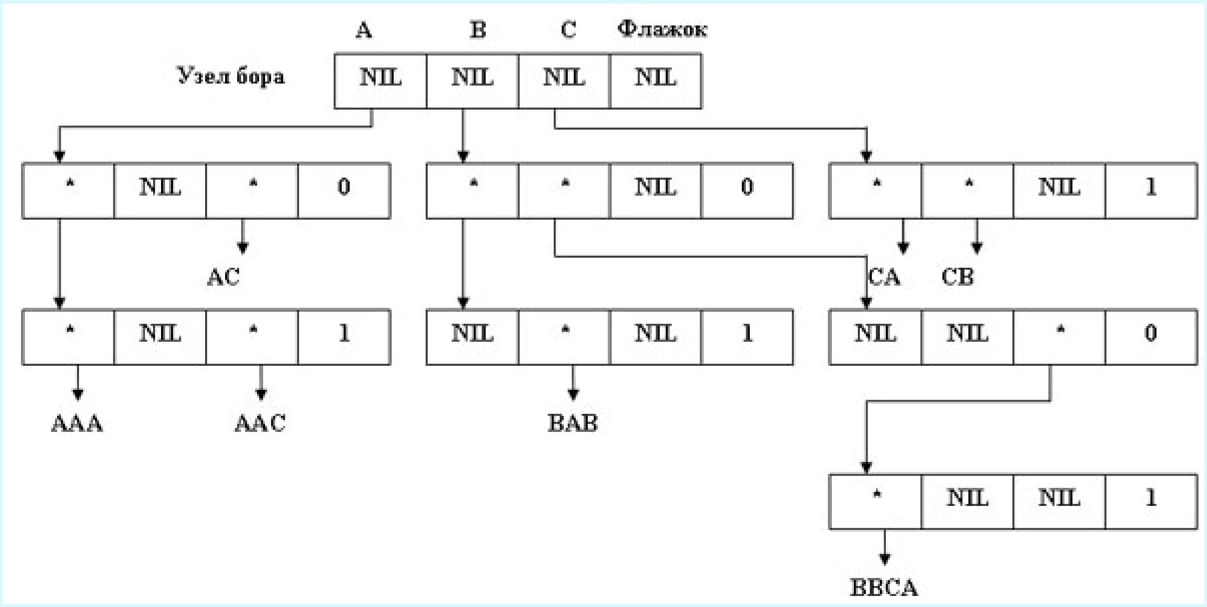


Fig.3. Image of **TRIE** structure

Although each node has room for **V** possible values, many of the value combinations are not used as key identifiers. This can result in large amounts of wasted memory. Clearly, if a particular node position is not used, its pointers are set to **NIL**, reflecting the fact that the subtree corresponding to the prefix is empty.

Example 3. Implementation of **a TRIE** tree in **LISP**.

```
(DEFUN MAIN (LAMBDA NIL
 ; Dictionary construction and search in it ;
   (PRIN1 "Input first letter... ")
   (PRINT (SETQ WORD1 (LIST (READ))))
   (LOOP
      (PRIN1 "Input keyword: ") (SETQ WORD2 (READ))
      ( (EQ WORD2 END) 'BYE-BYE )
      ; The & symbol marks the end of a word WORD2 ;
      (SETQ WORD2 (APPEND (UNPACK WORD2) (LIST '&)))
      (SETQ WORD1 (INSERT WORD2 WORD1))
      (PRINT WORD1)
   )
   (LOOP
      (PRIN1 "Input keyword for search: ")
      (SETQ WORD2 (READ))
      ( (EQ WORD2 END) 'BYE-BYE )
      (SETQ WORD2 (APPEND (UNPACK WORD2) (LIST '&)))
      (PRIN1 "Result: ") (PRINT (SEARCH WORD2 WORD1))
   )
))
; ----------------------------- ;
(DEFUN SEARCH (LAMBDA (WORD TRIE)
; Search for the word WORD in the TRIE dictionary ;
   (COND ( (NULL TRIE) NIL )
         ( (EQUAL WORD TRIE) T )
         ( (EQ (CAR WORD) '&) "Subword" )
         ( (MEMBER (CAR WORD) TRIE)
             (SEARCH (CDR WORD)
                     (CADR (MEMBER (CAR WORD) TRIE)))
         )
         (  T  NIL )
   )
))
; ---------------------------- ;
(DEFUN INSERT (LAMBDA (WORD LST)
; Putting the word WORD into the LST dictionary ;
   (COND ( (NULL LST) (TREE WORD) )
         ( (NULL WORD) LST )
         ( (NOT (MEMBER (CAR WORD) LST))
                   (APPEND LST (TREE WORD)) )
         ; -------------------------------- ;
         ( (AND (EQ (CAR LST) (CAR WORD))
                (ATOM (CADR LST))
                (NULL (CDR WORD))
           )
                   (CONS (CAR LST) (CDR LST))
         )
         ; ---------------------------- ;
         ( (AND (EQ (CAR LST) (CAR WORD))
                (ATOM (CADR LST))
           )
                   (CONS (CAR LST)
                         (APPEND (LIST (TREE (CDR WORD)))
                                 (CDR LST)))
         )
         ; ---------------------------- ;
         ( (AND (EQ (CAR LST) (CAR WORD))
                (NOT (ATOM (CADR LST))))
```

```
                      (CONS (CAR LST)
                            (APPEND
                               (LIST (INSERT (CDR WORD)
                                             (CADR LST)))
                               (CDDR LST)))
            )
            ; ----------------------------- ;
            ( (NOT (EQ (CAR LST) (CAR WORD)))
                    (CONS (CAR LST) (INSERT WORD (CDR LST)))
            )
      )
   ))
   ; --------------------- ;
   (DEFUN TREE (LAMBDA (LST)
   ; Construction from list (A B C...) of list ;
   ; with the following structure: (A (B (C (...)))) ;
      (COND ( (NULL (CDR LST)) (LIST (CAR LST)) )
            (    T   (LIST (CAR LST) (TREE (CDR LST))) )
      )
   ))
   ; ----------------------- ;
   (DEFUN APPEND (LAMBDA (X Y)
   ; The APPEND function returns a list consisting of ;
   ; the elements of LST1 appended to LST2 ;
       (COND ( (NULL X) Y )
             (  T  (CONS (CAR X) (APPEND (CDR X) Y)) )
       )
   ))
```

Test example:

```
    $ Input first letter... A
    (A)
    $ Input keyword: ASD
    (A (S (D (&))))
    $ Input keyword: ASE
    (A (S (D (&) E (&))))
    $ Input keyword: AKE
    $ (A (S (D (&) E (&)) K (E (&))))
    $ Input keyword: AKK
    $ (A (S (D (&) E (&)) K (E (&) K (&))))
    $ Input keyword: END
    $ Input keyword for search: ASE
    $ Result: T
    $ Input keyword for search: AS
    $ Result: Subword
    $ Input keyword for search: E
    $ Result: NIL
    $ Input keyword for search: END
```

TRIE trees are data structures that are as efficient as hashing methods.

Notes.

1. *The literature suggests many hybrid tree structures that combine the best qualities of the hashing method, binary trees,* ***B*** *-trees and their numerous variants. Among them, the monograph [1, p.134] names:* ***hash trees, extended hashing method, virtual B-tree****.*
2. *Using* ***TRIE*** *trees, one can construct positional trees [3], which are often used when searching for a substring in a string. A stronger result on recognizing the occurrence of one chain in another was obtained* ***by Yu. V. Matiyasevich*** *in 1969 (see his article "On the recognition of the occurrence relation in real time". Proceedings of Scientific Seminars of LOMI, v. 20, 104-114).*

[1] Tiori T., Fry J. Design of database structures: In 2 books. Book 2 - Moscow: Mir, 1985. - 320 p.
[2] Knuth D. The art of computer programming. V. 3: Sorting and searching. - Moscow: Mir, 1978. - 844 p.
[3] Aho A., Hopcroft J., Ullman J. Construction and analysis of computational algorithms. - Moscow: Mir, 1979. - 536 p.

From the next step we will start to consider ***the simplest algorithms on graphs***.

Step 77.
Implementation of the simplest algorithms on graphs. General information

In this step we will provide ***general information on the implementation of algorithms on graphs***.

When writing this section, the results presented in the monograph [1, Chapter 4] were significantly used.

Getting to know the methods of representing and processing graphs is very instructive. On the one hand, graphs are quite visual objects. On the other hand, as we will soon see, the machine representation of graphs allows for a great variety. The complexity of obtaining an answer to a particular question regarding a given graph depends, naturally, ***on the method of representing the graph***. Therefore, in graph algorithms, the relationship "algorithm - data structure" is very strong. The same algorithm, implemented on different data structures, often leads to completely different programs. This fact will allow us to illustrate the broad capabilities of the **LISP** language using a relatively small number of algorithms.

Graph algorithms also have great ***practical significance*** in computer science. First of all, we should mention the use of graphs in electronics [2]. Although the algorithms and implementations that we will describe in this section are hardly suitable for solving practically important problems, we still hope that the reader interested in the applied aspects of programming in the **LISP** language will get an idea of the advantages and disadvantages of this language.

Since our main goal is to introduce programming methods in **LISP**, all definitions related to graph theory will be given only to the extent necessary for the presentation. Additional information on graph theory can be found in books [3-12]. As an introduction to combinatorial algorithms, we can recommend [4, 7, 8, 11].

[1] Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.
[2] Ulman J. Computational aspects of SBIS. - M.: Radio and communication, 1990. - 480 p.
[3] Papadimitriou H, Steinglitz K. Combinatorial optimization. - M.: Mir, 1985. - 512 s.
[4] Lipsky V. Combinatorics for Programmers: Trans. with Polish. - M.: Mir, 1988. - 213 s.
[5] Kasyanov V.N., Sabelfeld V.K. Collection of tasks on the workshop on the computer. - M.: Nauka, 1986. - 272 s.
[6] Evstigneev V.A. Application of graph theory in programming. - M.: Nauka, 1985. - 352 s.
[7] Ore O. Graphs and their applications. - M.: Mir, 1965.- 174 s.
[8] Harari F. Graph theory. - M.: Mir, 1973. - 300 s.
[9] Zykov AA Fundamentals of graph theory. M.: Nauka, 1987. - 384 s.
[10] Lectures on the theory of graphs / Emelichev VA, Melnikov OI, Sarvanov VI, Tyshkevich RI - M.: Nauka, 1990. - 384 s.
[11] Wilson R. Introduction to graph theory. - M.: Mir, 1977. - 207 s.
[12] Reingold E, Nievergelt Y, Deo N. Combinatorial algorithms. Theory and practice. - M.: Mir, 1980. - 476 s.

In the next step we will provide ***some general information about graphs***.

Step 78.
Some concepts of graph theory

In this step we will introduce ***some concepts of graph theory***.

Regarding the terminology in graph theory, it is appropriate to quote from the book of the authoritative specialist F. Harari [1, p. 21]: "Most specialists in graph theory use their own terminology in books, articles and lectures... Even the word "graph" itself is not sacred. Some authors actually define "graph" as a graph, while others have in mind such concepts as a multigraph, pseudograph, directed graph or network. It seems to us that uniformity in the terminology of graph theory will never be achieved, but perhaps it is not necessary."

Let us first give ***an informal definition*** of a graph.

A directed graph is a pair **(V,E)**, where **V** is the set of vertices and **E** is the set of edges; the elements of **E** are ***ordered*** pairs of vertices.

An undirected graph is a pair **(V,E)**, where **V** is the set of vertices and **E** is the set of edges; the elements of the set E are ***unordered*** pairs of vertices.

The formal definition of an undirected graph is as follows. Let **V** be a non-empty set, **V2** be the set of all its two-element subsets. The pair **(V,E)**, where **E** is an arbitrary subset **of V2**, is called ***an undirected graph***.

Let us now give ***a formal definition of a directed graph***. Let **V** be a non-empty set, **VxV** its Cartesian square. The pair **(V,A)**, where **A** is an arbitrary subset **of VxV**, is called ***a directed graph***.

The elements of the set **A** are called ***directed edges*** or ***arcs***. If **(v_1, v_2)** is an arc, then the vertices **v_1** and **v_2** are called its ***beginning*** and ***end***, respectively.

A graph **(V,E)** can be viewed as ***a two-place (binary) relation*** **E** on a set **V**, which is defined as the set of ordered (or unordered) pairs of elements of the set **V**.

When speaking about binary relations as graphs, we simply mean the possibility of their visual representation and the use of methods and terminology characteristic of graphs. ***A graph*** is usually depicted on a plane as ***a set of points*** corresponding to vertices and ***lines*** connecting them corresponding to edges. Vertices can be located on the plane in an arbitrary manner. Moreover, it does not matter whether the edge connecting two vertices is a segment of a straight line or a curved line, since the fact of connecting two given vertices by an edge is important.

If it is necessary to distinguish the vertices of a graph, they are numbered or designated by different letters. In the graph representation on a plane, there may be points of intersection of edges that are not vertices. Below, in order to distinguish vertices from such points, we will depict vertices ***as circles***.

An example of a geometric representation of a graph is a subway line diagram.

The number of vertices of a graph **G** is called its ***order***. A graph of order **N** is called ***labeled*** if its vertices are assigned some labels, for example, numbers 1, 2, ..., **N**.

In the case of an undirected graph, two vertices between which there is an edge are said to be ***adjacent***. More formally, two vertices **u** and **v** of a graph are said to be ***adjacent*** if the set **{u, v}** is an edge, and ***nonadjacent*** otherwise.

We call two edges ***adjacent*** if they have a common vertex.

A vertex **v** and an edge **e** are called ***incident*** if **v** is an endpoint of edge **e** (i.e. **e=uv**), and ***non-incident*** otherwise.

Note that adjacency is a relationship between homogeneous elements of a graph, whereas incidence is a relationship between heterogeneous elements.

A simple path in a graph is a sequence of adjacent edges (V_1,V_2) (V_2,V_3)... (V_{k-1},V_k) such that all V_i except possibly V_1 and V_k are distinct. Note that for a directed graph this definition means that the directions of the edges along the entire path ***are consistent***.

The number of edges that make up a path is called its ***length***.

A simple path that starts and ends at the same vertex is called ***a cycle***.

An undirected graph is called ***connected*** if any two of its vertices can be connected by a path.

A directed graph ***is connected*** if the graph obtained from it by removing the orientation of its edges is connected.

In a connected undirected graph, we can introduce the concept of ***distance between vertices***. ***The distance between vertices*** **U** and **V** is defined as ***the minimum length of the path*** between **U** and **V**.

A graph **H** is called ***a subgraph of*** a graph **G** if all vertices and edges of **H** are at the same time vertices and edges of **G**. In this case, we also say that **H** ***is contained in*** **G**.

Consider some family of subgraphs of graph **G.** A graph H_{max} from this family is called ***maximal*** if it is not contained in any other graph from the family under consideration.

An undirected connected graph is called ***a tree*** if it contains no cycles.

A tree with a selected vertex is called ***a rooted tree***, and the selected vertex itself is called ***the root***.

A maximal tree subgraph is called ***a spanning tree of a graph***, and a maximal connected subgraph is called ***a connected component*** of a graph. A connected graph has a unique connected component, which coincides with the graph itself.

Two graphs **G** and **H** ***are isomorphic*** if there exists a one-to-one mapping (called ***an isomorphism***) from the vertex set of **G** to the vertex set of **H** that preserves adjacency. ***An automorphism*** of **G** is an isomorphism of **G** onto itself.

---

(1) Harari F. Graph Theory. - M.: Mir, 1973. - 300 p.

In the next step we will start looking at ***ways to represent graphs***.

Step 79.
LISP and graph representations. Adjacency matrix

In this step, we will look at ***the application of the adjacency matrix to represent graphs***.

Let **G** be ***a labeled graph*** of order **N**, **V = {1, 2, ..., N}**.

The best way to represent a graph is as ***an adjacency matrix***, defined as a matrix **B = [B_{ij}]** of size **NxN**, where **B_{ij} =1** if there is an edge from vertex **i** to vertex **j**, and **B_{ij} =0** otherwise. Note that the number of ones in a row of an adjacency matrix is equal to the degree of the corresponding vertex.

It is easy to see that the adjacency matrix of an undirected graph ***is always symmetric***. This is not the case for a directed graph.

The main ***advantage of*** the adjacency matrix is that in one step one can get an answer to the question "is there an edge from **i** to **j** ?".

The disadvantage is the fact that regardless of the number of edges, the amount of memory occupied is **NxN**. In practice, this inconvenience can sometimes be reduced by storing an entire row (column) of the matrix in one machine word - this is possible for small **N**.

To represent a matrix as an **S** -expression, we use the following structure: we define the matrix rows as ordered lists of matrix elements, and the matrix itself as a list of rows.

Let us give an example of constructing an **S** -expression corresponding to a directed graph:

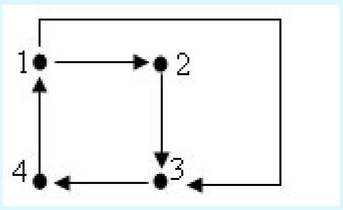


Fig. 1. Directed graph

```
             Matrix S-expression
            adjacencies
           (0 1 1 0)           ( (0 1 1 0)
           (0 0 1 0)             (0 0 1 0)
           (0 0 0 1)             (0 0 0 1)
           (1 0 0 0)             (1 0 0 0)
                               )
```

In an undirected graph **GRAPH,** we find a list of neighbors of a vertex **X** (i.e. vertices connected to it by at least one edge) using the graph representation as ***an adjacency matrix***.

Program.

```
(DEFUN NEIGHBOUR1 (LAMBDA (X GRAPH)
   (VSPOM 1 (NTH X GRAPH))
))
; ------------------------- ;
(DEFUN VSPOM (LAMBDA (I LST)
   (COND ( (NULL LST) NIL )
         ( (EQ (CAR LST) 1)
              (CONS I (VSPOM (+ I 1) (CDR LST))) )
         (  T  (VSPOM (+ I 1) (CDR LST)) )
   )
))
; ----------------------- ;
(DEFUN NTH (LAMBDA (N LST)
; The function returns the N-th element of the list LST ;
   (COND ( (EQ N 1) (CAR LST) )
         (  T  (NTH (- N 1) (CDR LST)) )
   )
))
```

Test examples:

```
   $ (NEIGHBOUR1 2 '((0 1 1 0) (0 0 1 0) (0 0 0 1) (1 0 0 0)))
   (3)
   $ (NEIGHBOUR1 1 '((0 1 1 0) (0 0 1 0) (0 0 0 1) (1 0 0 0)))
   (2 3)
   $ (NEIGHBOUR1 4 '((0 1 1 0) (0 0 1 0) (0 0 0 1) (1 0 0 0)))
   (1)
```

It is quite obvious that the implementation of the adjacency matrix ***using* the S- *expression*** considered in the program is very inefficient. Its main drawback is the lack of ***direct access to the matrix elements*** (in some dialects of the **LISP** language). In the following steps of the section, we will improve this representation!

***Notes*.**

1. *Graphs **are isomorphic** if and only if their **adjacency matrices** are obtained from each other by identical permutations of rows and columns [1, p.28].*
2. *In graph theory, the classical way of representing a graph is **the incidence matrix** [2, pp. 84-85]. This is a matrix **A** with **N** rows corresponding to the vertices and **M** columns corresponding to the edges.*

   *For **a directed graph**, the column corresponding to an arc **<x,y>** belonging to **E** contains -1 in the row corresponding to vertex **x**, 1 in the row corresponding to vertex **y**, and zeros in all other rows (**a loop**, i.e. an arc of the form **<x,x>**, is conveniently represented by another value in row **x**, for example, 2).*

   *In the case of **an undirected** graph, the column corresponding to the edge **{x,y}** contains 1 in the rows corresponding to **x** and **y**, and zeros in the remaining rows.*

   *From an algorithmic point of view, the incidence matrix is probably **the worst way to represent a graph** that one can imagine.*

   *This method requires **NxM** memory cells, and most of these cells are occupied by zeros. Access to information is also inconvenient. Answering elementary questions like "does the arc **<x,y> exist?", "to which vertices do the edges from x** lead ?" requires in the worst case to enumerate all the columns of the matrix, and therefore **M** steps.*

3. *Graphs (directed graphs) **are isomorphic** if and only if their **incidence matrices** are obtained from each other by arbitrary permutations of rows and columns [1, p.28].*
4. *There are several other ways to represent graphs using matrices:*
   - ***reachability matrix** (Warshall's algorithm is used to construct it) [3, p.439],*
   - ***matrix of cycles** [4, p.183-186],*
   - ***Kirchhoff matrix** [1, p.30-31].*

(1) Lectures on Graph Theory / Emelichev V.A., Melnikov O.I., Sarvanov V.I., Tyshkevich R.I. - M.: Nauka, 1990. - 384 p.
(2) Lipskiy V. Combinatorics for Programmers: Transl. from Polish. - M.: Mir, 1988. - 213 p.
(3) Traamble J., Sorenson P. Introduction to Data Structures. - M.: Mashinostroenie, 1982. - 784 p.
(4) Harari F. Graph Theory. - M.: Mir, 1973. - 300 p.

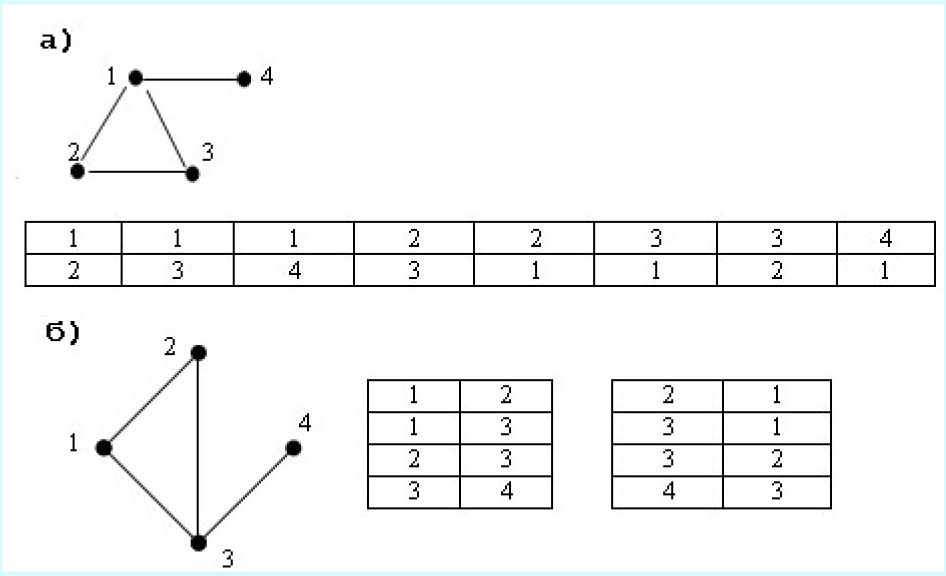
In the next step we will look at ***using adjacency structures to represent graphs***.

Step 80.
LISP and graph representations. Adjacency structures

In this step we will look at ***using adjacency structures to represent graphs***.

More economical in terms of memory (especially in the case of ***non-dense graphs***, when **M** is much smaller than **NxN**) is the method of representing the graph using the so-called ***adjacency structure***, which is, in the simplest case, ***a list of pairs corresponding to its edges***. The pair **<x,y>** corresponds to the arc **<x,y>** if the graph is directed, and to the edge **{x,y}** in the case of an undirected graph.

For example, let us give lists of edges corresponding to graphs:



1	1	1	2	2	3	3	4
2	3	4	3	1	1	2	1

1	2
1	3
2	3
3	4

2	1
3	1
3	2
4	3

Fig.1. Edge lists

Obviously, the memory footprint in this case is **2xM**. The disadvantage is the large number of steps (of the order of M in the worst case) required to obtain the set of vertices to which edges from a given vertex lead.

The situation can be improved considerably by ordering the set of pairs ***lexicographically*** and using binary search, but a better solution in many cases is a data structure that we will call ***incidence lists***. It contains, for each node **v** in **V,** a list of nodes adjacent to it. More precisely, each element of such a list is a record **R** containing a ***node*** **R.String** and a pointer **R.Next** ***to*** the next entry in the list (**R.Next = Nil** for the last entry in the list).

The pointer ***START*** **[v]** is a pointer to the beginning of a list containing vertices from the set of vertices adjacent to a given vertex **v**.

Note that for undirected graphs, each edge is represented ***twice*** in the incidence lists.

Let us give an example of representing an undirected graph using incidence lists (**N** denotes **Nil**):

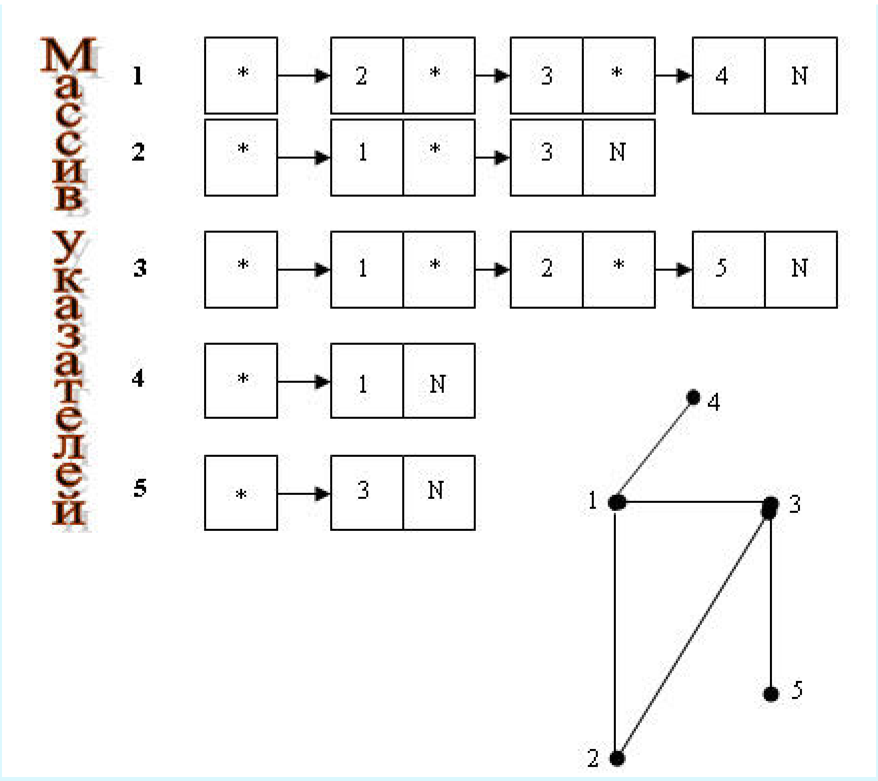


Fig.2. Representation of an undirected graph

In many algorithms, the structure of the graph is dynamically modified by adding and deleting edges. In such cases, we assume that in our incidence lists, the element of the list ***START*** **[u]** containing the node **v** is provided with a pointer to the element of the list ***START*** **[v]** containing the node **u**, and that each element of the list contains a pointer not only to the next element but also to the previous one. Then, when deleting some element from the list, we can easily, in a constant number of steps, delete another element representing the same edge without traversing the list containing this element.

The number of memory cells required to represent a graph using incidence lists will obviously be of the order **M+N**.

Representing the adjacency structure as ***a list of edges*** is the simplest way to represent a graph in **LISP**. It is natural to implement such a list as ***a list of dotted pairs of adjacent vertices***. However, with this approach, special care must be taken to deal with vertices that are not connected to other vertices by edges. For example, to represent such "lonely" vertices, one can add to the list of edges a fictitious edge connecting the "lonely" vertex with the **FALSE** vertex.

As examples of using the method of representing a graph as ***a list of edges,*** we will construct:

- a function that, in ***a directed graph*** **GRAPH,** finds a list ***of neighbors of a vertex*** **X** (i.e. vertices connected to it by at least one edge):

```
  (DEFUN NEIGHBOUR2 (LAMBDA (X GRAPH)
     (COND ( (NULL GRAPH) NIL )
           ( (AND (EQ (CAAR GRAPH) X)
                  (EQ (CDAR GRAPH) FALSE)) NIL )
           ( (EQ (CAAR GRAPH) X)
                  (CONS (CDAR GRAPH)
                        (NEIGHBOUR2 X (CDR GRAPH))) )
           ( (EQ (CDAR GRAPH) X)
                  (CONS (CAAR GRAPH)
                        (NEIGHBOUR2 X (CDR GRAPH))) )
           (  T  (NEIGHBOUR2 X (CDR GRAPH)) )
     )
```

```
))
```

Test examples:

```
$ (NEIGHBOUR2 2 '((1 . 2) (1 . 3) (2 . 3) (5 . FALSE)))
(1 3)
$ (NEIGHBOUR2 1 '((1 . 2) (1 . 3) (2 . 3) (5 . FALSE)))
(2 3)
$ (NEIGHBOUR2 5 '((1 . 2) (1 . 3) (2 . 3) (5 . FALSE)))
NIL
```

- a function that, in ***an undirected graph* GRAPH,** finds a list ***of neighbors of a vertex* X** (i.e. vertices connected to it by at least one edge):

```
(DEFUN MAIN (LAMBDA (X GRAPH)
   (LIST-SET (NEIGHBOUR2 X GRAPH))
))
; ------------------------------- ;
(DEFUN NEIGHBOUR2 (LAMBDA (X GRAPH)
   (COND ( (NULL GRAPH) NIL )
         ( (AND (EQ (CAAR GRAPH) X)
                (EQ (CDAR GRAPH) 'FALSE)) NIL )
         ( (EQ (CAAR GRAPH) X)
                (CONS (CDAR GRAPH)
                      (NEIGHBOUR2 X (CDR GRAPH))) )
         ( (EQ (CDAR GRAPH) X)
                (CONS (CAAR GRAPH)
                      (NEIGHBOUR2 X (CDR GRAPH))) )
         (  T  (NEIGHBOUR2 X (CDR GRAPH)) )
   )
))
; ------------------------- ;
(DEFUN LIST-SET (LAMBDA (LST)
; The LIST-SET function converts an list LST into a list in which
each element occurs only once ;
   (COND ( (NULL LST) NIL )
         ( (MEMBER (CAR LST) (CDR LST))
             (LIST-SET (CDR LST)) )
         ( T (CONS (CAR LST)
                   (LIST-SET (CDR LST))) )
   )
))
```

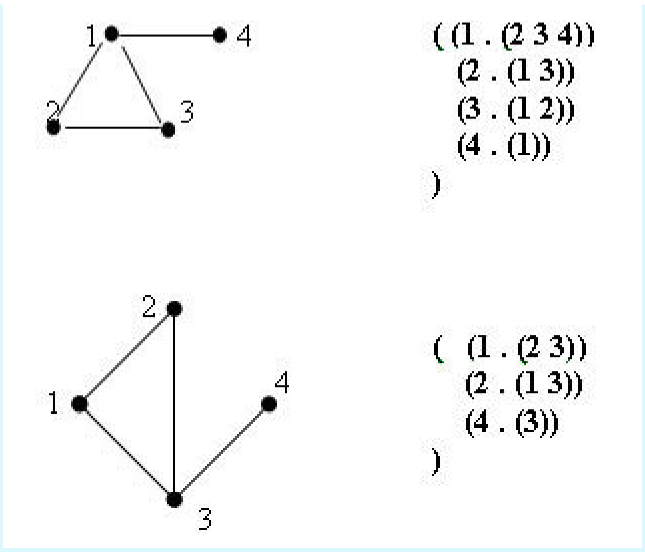
Test examples:

```
$ (MAIN 2 '((1 . 2) (1 . 3) (2 . 3) (2 . 1) (3 . 1) (3 . 2) (5 . FALSE)))
(1 3)
$ (MAIN 1 '((1 . 2) (1 . 3) (2 . 3) (2 . 1) (3 . 1) (3 . 2) (5 . FALSE)))
(2 3)
$ (MAIN 5 '((1 . 2) (1 . 3) (2 . 3) (2 . 1) (3 . 1) (3 . 2) (5 . FALSE)))
NIL
```

It is clear that when a graph is represented as a list of edges, information about the vertices may be difficult to access. This will be the case when the number of edges is much greater than the number of vertices.

The most convenient representation of graphs in the **LISP** language is the representation of the adjacency structure as ***an associative list***. The first element of a dotted pair included in this list is a graph vertex. The second element is a list of vertices adjacent to the given one.

Let us give examples of representing the adjacency structure in the form of an associative list:



```
((1 . (2 3 4))
  (2 . (1 3))
  (3 . (1 2))
  (4 . (1))
)

(  (1 . (2 3))
   (2 . (1 3))
   (4 . (3))
)
```

Fig.3. Representation as an associative list

Now the problem of finding all the neighbors of a given vertex is solved quite simply [1, p.127]:

```
(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
   (COND ( (NULL (ASSOC X GRAPH)) NIL )
         (  T   (CDR (ASSOC X GRAPH)) )
   )
))
```

Test examples:

```
$ (NEIGHBOUR3 2 '((1 . (2 3 4)) (2 . (1 3)) (3 . (1 2)) (4 . (1))))
(1 3)
$ (NEIGHBOUR3 1 '((1 . (2 3 4)) (2 . (1 3)) (3 . (1 2)) (4 . (1))))
(2 3 4)
$ (NEIGHBOUR3 3 '((1 . (2 3 4)) (2 . (1 3)) (3 . (1 2)) (4 . (1))))
(1 2)
```

(1) Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.

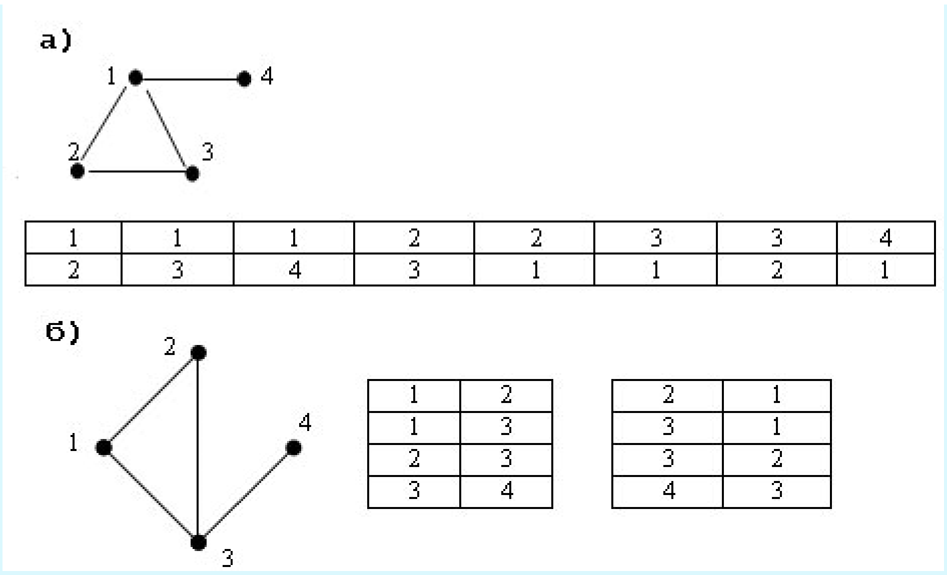
In the next step we will look at ***representing graphs using dynamic structures***.

Step 80.
LISP and graph representations. Adjacency structures

In this step we will look at ***using adjacency structures to represent graphs***.

More economical in terms of memory (especially in the case of ***non-dense graphs***, when **M** is much smaller than **NxN**) is the method of representing the graph using the so-called ***adjacency structure***, which is, in the simplest case, ***a list of pairs corresponding to its edges***. The pair **<x,y>** corresponds to the arc **<x,y>** if the graph is directed, and to the edge **{x,y}** in the case of an undirected graph.

For example, let us give lists of edges corresponding to graphs:



1	1	1	2	2	3	3	4
2	3	4	3	1	1	2	1

1	2
1	3
2	3
3	4

2	1
3	1
3	2
4	3

Fig.1. Edge lists

Obviously, the memory footprint in this case is **2xM**. The disadvantage is the large number of steps (of the order of M in the worst case) required to obtain the set of vertices to which edges from a given vertex lead.

The situation can be improved considerably by ordering the set of pairs ***lexicographically*** and using binary search, but a better solution in many cases is a data structure that we will call ***incidence lists***. It contains, for each node **v** in **V,** a list of nodes adjacent to it. More precisely, each element of such a list is a record **R** containing a ***node*** **R.String** and a pointer **R.Next** ***to*** the next entry in the list (**R.Next = Nil** for the last entry in the list).

The pointer ***START*** **[v]** is a pointer to the beginning of a list containing vertices from the set of vertices adjacent to a given vertex **v**.

Note that for undirected graphs, each edge is represented ***twice*** in the incidence lists.

Let us give an example of representing an undirected graph using incidence lists (**N** denotes **Nil**):

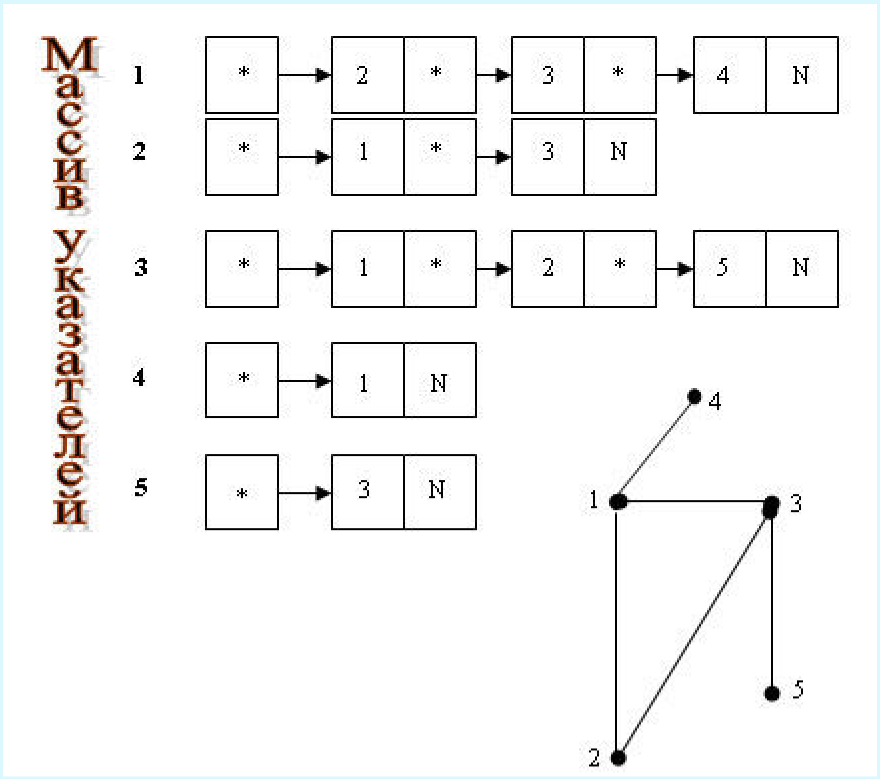


Fig.2. Representation of an undirected graph

In many algorithms, the structure of the graph is dynamically modified by adding and deleting edges. In such cases, we assume that in our incidence lists, the element of the list ***START*** **[u]** containing the node **v** is provided with a pointer to the element of the list ***START*** **[v]** containing the node **u**, and that each element of the list contains a pointer not only to the next element but also to the previous one. Then, when deleting some element from the list, we can easily, in a constant number of steps, delete another element representing the same edge without traversing the list containing this element.

The number of memory cells required to represent a graph using incidence lists will obviously be of the order **M+N**.

Representing the adjacency structure as ***a list of edges*** is the simplest way to represent a graph in **LISP**. It is natural to implement such a list as ***a list of dotted pairs of adjacent vertices***. However, with this approach, special care must be taken to deal with vertices that are not connected to other vertices by edges. For example, to represent such "lonely" vertices, one can add to the list of edges a fictitious edge connecting the "lonely" vertex with the **FALSE** vertex.

As examples of using the method of representing a graph as ***a list of edges,*** we will construct:

- a function that, in ***a directed graph*** **GRAPH,** finds a list ***of neighbors of a vertex*** **X** (i.e. vertices connected to it by at least one edge):

```
(DEFUN NEIGHBOUR2 (LAMBDA (X GRAPH)
   (COND ( (NULL GRAPH) NIL )
         ( (AND (EQ (CAAR GRAPH) X)
                (EQ (CDAR GRAPH) FALSE)) NIL )
         ( (EQ (CAAR GRAPH) X)
                (CONS (CDAR GRAPH)
                      (NEIGHBOUR2 X (CDR GRAPH))) )
         ( (EQ (CDAR GRAPH) X)
                (CONS (CAAR GRAPH)
                      (NEIGHBOUR2 X (CDR GRAPH))) )
         (  T   (NEIGHBOUR2 X (CDR GRAPH)) )
   )
))
```

Test examples:

```
$ (NEIGHBOUR2 2 '((1 . 2) (1 . 3) (2 . 3) (5 . FALSE)))
(1 3)
$ (NEIGHBOUR2 1 '((1 . 2) (1 . 3) (2 . 3) (5 . FALSE)))
(2 3)
$ (NEIGHBOUR2 5 '((1 . 2) (1 . 3) (2 . 3) (5 . FALSE)))
NIL
```

- a function that, in ***an undirected graph*** **GRAPH,** finds a list ***of neighbors of a vertex*** **X** (i.e. vertices connected to it by at least one edge):

```
(DEFUN MAIN (LAMBDA (X GRAPH)
   (LIST-SET (NEIGHBOUR2 X GRAPH))
))
; ------------------------------- ;
(DEFUN NEIGHBOUR2 (LAMBDA (X GRAPH)
   (COND ( (NULL GRAPH) NIL )
         ( (AND (EQ (CAAR GRAPH) X)
                (EQ (CDAR GRAPH) 'FALSE)) NIL )
         ( (EQ (CAAR GRAPH) X)
                (CONS (CDAR GRAPH)
                      (NEIGHBOUR2 X (CDR GRAPH))) )
         ( (EQ (CDAR GRAPH) X)
                (CONS (CAAR GRAPH)
                      (NEIGHBOUR2 X (CDR GRAPH))) )
         (  T  (NEIGHBOUR2 X (CDR GRAPH)) )
   )
))
; ------------------------- ;
(DEFUN LIST-SET (LAMBDA (LST)
; The LIST-SET function converts an list LST into a list in which
each element occurs only once ;
   (COND ( (NULL LST) NIL )
         ( (MEMBER (CAR LST) (CDR LST))
             (LIST-SET (CDR LST)) )
         ( T (CONS (CAR LST)
                   (LIST-SET (CDR LST))) )
   )
))
```

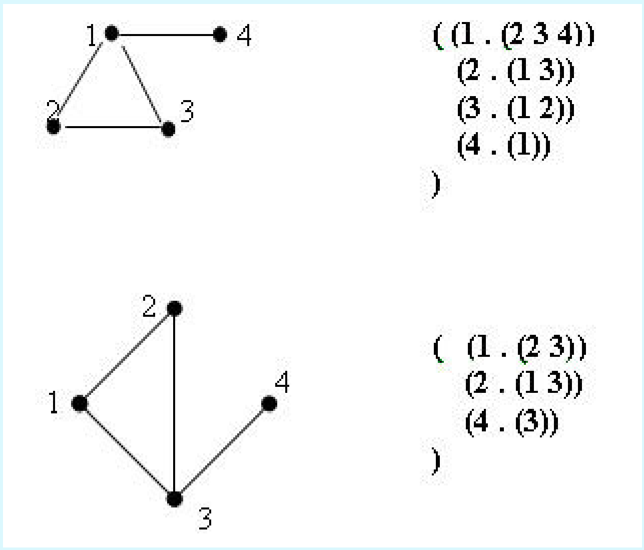
Test examples:

```
   $ (MAIN 2 '((1 . 2) (1 . 3) (2 . 3) (2 . 1) (3 . 1) (3 . 2) (5 . FALSE)))
   (1 3)
   $ (MAIN 1 '((1 . 2) (1 . 3) (2 . 3) (2 . 1) (3 . 1) (3 . 2) (5 . FALSE)))
   (2 3)
   $ (MAIN 5 '((1 . 2) (1 . 3) (2 . 3) (2 . 1) (3 . 1) (3 . 2) (5 . FALSE)))
   NIL
```

It is clear that when a graph is represented as a list of edges, information about the vertices may be difficult to access. This will be the case when the number of edges is much greater than the number of vertices.

The most convenient representation of graphs in the **LISP** language is the representation of the adjacency structure as ***an associative list***. The first element of a dotted pair included in this list is a graph vertex. The second element is a list of vertices adjacent to the given one.

Let us give examples of representing the adjacency structure in the form of an associative list:



```
((1 . (2 3 4))
  (2 . (1 3))
  (3 . (1 2))
  (4 . (1))
)
```

```
(  (1 . (2 3))
   (2 . (1 3))
   (4 . (3))
)
```

Fig.3. Representation as an associative list

Now the problem of finding all the neighbors of a given vertex is solved quite simply [1, p.127]:

```
   (DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
      (COND ( (NULL (ASSOC X GRAPH)) NIL )
            (  T  (CDR (ASSOC X GRAPH)) )
      )
   ))
```

Test examples:

```
   $ (NEIGHBOUR3 2 '((1 . (2 3 4)) (2 . (1 3)) (3 . (1 2)) (4 . (1))))
   (1 3)
   $ (NEIGHBOUR3 1 '((1 . (2 3 4)) (2 . (1 3)) (3 . (1 2)) (4 . (1))))
   (2 3 4)
   $ (NEIGHBOUR3 3 '((1 . (2 3 4)) (2 . (1 3)) (3 . (1 2)) (4 . (1))))
   (1 2)
```

---

(1) Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.

In the next step we will look at ***representing graphs using dynamic structures***.

Step 81.
LISP and graph representations. Graph representation using dynamic structures

In this step we will look at ***the last way of representing graphs***.

Let us point out another more cumbersome way of representing graphs, namely: ***representing a graph using dynamic structures***.

For example, a directed graph and its memory representation can be depicted in a single figure as follows:

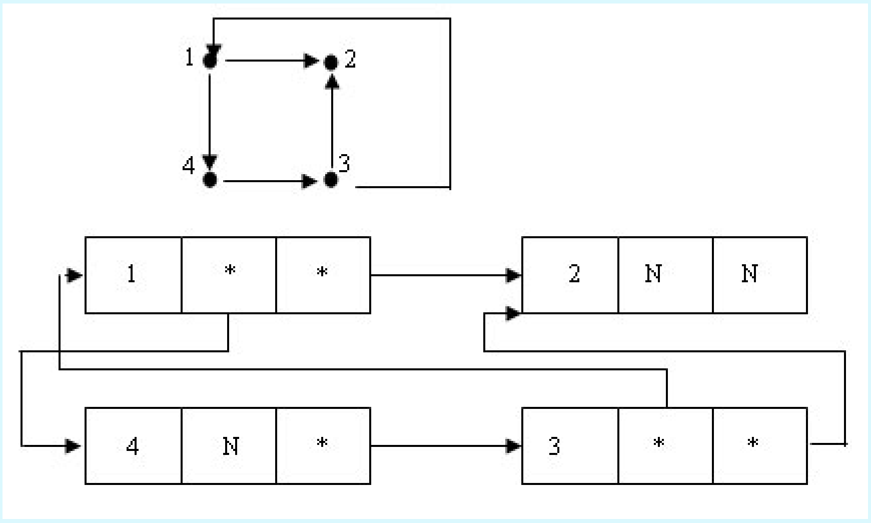


Fig. 1. Representation of a directed graph

By means of links, all connections in a structure that models a directed graph can be expressed: the graph vertices correspond to the dynamic variables themselves, and the edges correspond to links.

In a similar way, using a graph, one can represent ***syntactic diagrams and grammars*** [1, pp. 455-457].

---

(1) Tremblay J., Sorenson P. Introduction to data structures. - M.: Mashinostroenie, 1982. - 784 p.

In the next step we will start looking ***at graph search***.

Step 82.
Graph Search

In this step we will provide ***general information on organizing graph search in the* LISP** language.

One of the most common graph-related tasks is ***graph searching***.

By ***graph search*** we mean the process of systematically scanning all the vertices of a graph in order to find vertices that satisfy some condition.

To organize the simplest search in a graph defined using ***an unknown*** (!) list structure, the encapsulated data type ***set*** is used. The idea of the algorithm is clear from the program below.

Program.

```
(DEFUN WALK (LAMBDA (GRAPH)
   (COND ( (NULL GRAPH) NIL )
         ( T (LIST-SET (ONE-RANGE GRAPH)) )
   )
))
; ------------------------- ;
(DEFUN LIST-SET (LAMBDA (LST)
; The LIST-SET function converts an list LST into a set ;
   (COND ( (NULL LST) NIL )
         ( (MEMBER (CAR LST) (CDR LST))
             (LIST-SET (CDR LST)) )
         ( T (CONS (CAR LST)
                   (LIST-SET (CDR LST))) )
   )
))
; -------------------------- ;
(DEFUN ONE-RANGE (LAMBDA (LST)
; The list LST structure is "compressed" into ;
; a single-level list ;
   ( (NULL LST) NIL )
   ( (ATOM LST) (CONS (CAR LST) NIL) )
   ( APPEND (ONE-RANGE (CAR LST)) (ONE-RANGE (CDR LST)) )
))
```

Test example:

```
$ (WALK '((1 . (2 3 4)) (2 . (3)) (3 . (4))))
(1 2 3 4)
```

In the next step we will look ***at depth-first search***.

Step 83.
Depth-first search

In this step we will look at ***the depth-first search algorithm***.

The two most common graph search algorithms are called ***"depth-first search"*** and ***"breadth-first search"***. The first is based on such an order of examining the graph vertices, when a new vertex, if possible, is chosen among the neighbors of the current vertex. If among the neighbors of the current vertex there are none that have not been examined before, we return to the previous vertex and resume the search.

The program implementing this algorithm looks like this [1, p.128]:

```
(DEFUN DEPTHFIRST (LAMBDA (GRAPH ROOT)
 ; GRAPH - a graph specified as an adjacency structure ;
; ROOT - the graph vertex from which the search begins ;
; Result: a list of graph vertices in the order of traversal ;
   (COND ( (NULL GRAPH) NIL )
         (  T  (DEFI GRAPH (LIST ROOT) (LIST ROOT)) )
   )
))
; ------------------------------------ ;
(DEFUN DEFI (LAMBDA (GRAPH VISITED PATH)
; VISITED - a list of already viewed vertices ;
; PATH - a list of vertices defining the viewing path ;
   (COND ( (NULL PATH) (REVERSE VISITED) )
         ( T  (COND ( (NULL (EXPND GRAPH VISITED
                                    (CAR PATH)))
                      (DEFI GRAPH VISITED
                            (CDR PATH)) )
                    (  T  (DEFI GRAPH
                                (CONS (EXPND GRAPH VISITED
                                             (CAR PATH))
                                      VISITED)
                                (CONS (EXPND GRAPH VISITED
                                             (CAR PATH))
                                      PATH)) )) )
   )
))
; --------------------------------------- ;
(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)
; Select the next not yet visited vertex of the neighboring ;
; with VERTEX ;
   (COND ( (NULL (NEIGHBOUR3 VERTEX GRAPH)) NIL )
         (  T  (FIRSTNOTVISITED
                    VISITED
                    (NEIGHBOUR3 VERTEX GRAPH))
         )
   )
))
; ------------------------------------------ ;
(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)
   (COND ( (NULL VLIST) NIL )
         (  T  (COND ( (NULL (MEMBER (CAR VLIST) VISITED))
                          (CAR VLIST)
                     )
                     (  T  (FIRSTNOTVISITED
                                VISITED
                                (CDR VLIST)) )) )
   )
))
; ------------------------------ ;
(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
   (COND ( (NULL (ASSOC X GRAPH)) NIL )
         (  T  (CDR (ASSOC X GRAPH)) )
   )
))
```

Test examples:

```
$ (DEPTHFIRST '((1 . (2 3 4)) (2 . (3))   (3 . (4))) 1)
(1 2 3 4)
```

```
   $ (DEPTHFIRST '((1 . (2 3 4)) (2 . (3))   (3 . (4))) 2)
   (2 3 4)
   $ (DEPTHFIRST '((1 . (2 3 4)) (2 . (3))   (3 . (4))) 3)
   (3 4)
   $ (DEPTHFIRST '((1 . (2 3 4)) (2 . (3 1)) (3 . (4))) 2)
   (2 3 4 1)
   $ (DEPTHFIRST '((1 . (2 3)) (2 . (4 5))   (3 . (6 7))) 1)
   (1 2 4 5 3 6 7)
```

The last test example is illustrated by the following graph:

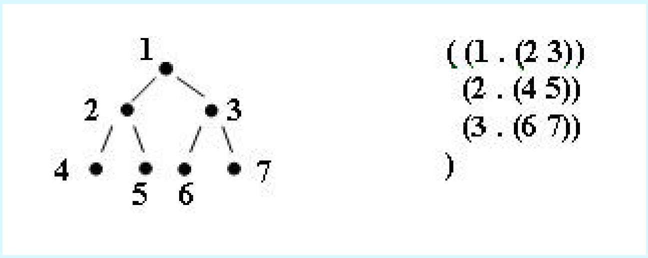


Fig. 1. Example of a graph

The program works as follows.

If the graph is not empty, then the first vertex of the graph is entered into two lists: **VISITED** - a list of already visited vertices and **PATH - a list of vertices defining the viewing path. After that, the DEFI** function is called. Its third argument - the **PATH** list - allows us to return to the previous vertex at any time.
The **VISITED** list is used to remember which vertices have already been visited. The selection of the next vertex is carried out using the **EXPND** function. It is the operation of this function that determines the order of viewing the graph. In our case, as long as possible, an adjacent vertex is selected for viewing, i.e. at each step of the algorithm an attempt is made to go deep into the graph. Otherwise, the first element is removed from the **PATH** list, and the search is resumed from the previous vertex.

The value of the DEPTHFIRST function is a list of the graph vertices in the order in which these vertices were examined. Obviously, this order depends on the vertex from which the examination begins. Moreover, for some graphs, the lists of examined vertices obtained for different values of the starting vertex do not have a single element in common.

We have already mentioned that the **PATH parameter of the DEFI** function describes the path from the initial vertex to the one being viewed. Therefore, the algorithm for finding a vertex can be easily modified into ***an algorithm for finding a path between specified vertices*** (for example, **the ROOT** and **END** vertices):

```
   (DEFUN WAY (LAMBDA (GRAPH ROOT END)
      (COND ( (NULL GRAPH) NIL )
            (  T  (DEFI GRAPH (LIST ROOT) (LIST ROOT) END) )
      )
   ))
   ; ---------------------------------------- ;
   (DEFUN DEFI (LAMBDA (GRAPH VISITED PATH END)
   ; VISITED - a list of already viewed vertices ;
   ; PATH - a list of vertices defining the viewing path ;
   ; END - the final vertex of the path ;
      (COND ( (NULL PATH) (REVERSE VISITED) )
            ( T  (COND ( (NULL (EXPND GRAPH VISITED (CAR PATH)))
                            (DEFI GRAPH VISITED (CDR PATH) END)
                       )
                       ( (EQ (EXPND GRAPH VISITED (CAR PATH)) END)
                                (REVERSE (CONS END PATH)) )
                       (  T  (DEFI GRAPH
                                   (CONS (EXPND GRAPH VISITED
                                                (CAR PATH))
                                   VISITED)
                             (CONS (EXPND GRAPH VISITED
                                          (CAR PATH))
                                   PATH)
                             END) )) )
   )
))
; --------------------------------------- ;
(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)
; Select the next unvisited vertex ;
; adjacent to VERTEX ;
   (COND ( (NULL (NEIGHBOUR3 VERTEX GRAPH)) NIL )
         (  T  (FIRSTNOTVISITED
                         VISITED
                         (NEIGHBOUR3 VERTEX GRAPH))
         )
   )
))
; ------------------------------------------ ;
(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)
   (COND ( (NULL VLIST) NIL )
         (  T  (COND ( (NULL (MEMBER (CAR VLIST) VISITED))
                         (CAR VLIST)
                     )
                     (  T  (FIRSTNOTVISITED
                                    VISITED
                                    (CDR VLIST))
                     )
               )
         )
   )
))
; ------------------------------- ;
(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
   (COND ( (NULL (ASSOC X GRAPH)) NIL )
         (  T  (CDR (ASSOC X GRAPH)) )
   )
))
```

Test examples:

```
$ (WAY '((1 . (2))(2 . (3 4))(3 . ())(4 . (5))(5 . (2 7 8))(7 . ())(8 . ())) 1 2)
(1 2)
$ (WAY '((1 . (2))(2 . (3 4))(3 . ())(4 . (5))(5 . (2 7 8))(7 . ())(8 . ())) 1 5)
(1 2 4 5)
$ (WAY '((1 . (2))(2 . (3 4))(3 . ())(4 . (5))(5 . (2 7 8))(7 . ())(8 . ())) 4 2)
(4 5 2)
$ (WAY '((1 . (2))(2 . (3 4))(3 . ())(4 . (5))(5 . (2 7 8))(7 . ())(8 . ())) 7 7)
(7)
$ (WAY '((1 . (2))(2 . (3 4))(3 . ())(4 . (5))(5 . (2 7 8))(7 . ())(8 . ())) 1 1)
(1 2 3 4 5 7 8)
```

Notes.

1. *The algorithm for depth-first search on a graph is described, for example, in the monographs [2, pp. 198-205], [3, pp. 88-91].*
2. ***In the formulation of a non-recursive depth-first search algorithm on a graph*** *given below [4, pp. 125-126], it is assumed, firstly, that some linear order is fixed on the set of all vertices of the graph, and, secondly, that the set of vertices adjacent to any vertex of the graph is also linearly ordered:*

```
WHILE   There is at least one unvisited node   DO
```

```
  BEGIN
     Let p   be the first (i.e. minimal) of the unvisited nodes
     vertices. Visit vertex p and place it in an empty
     stack S ;
      WHILE   Stack S is not empty   DO
        BEGIN
           Let p be the vertex on top of stack S ;
            IF Vertex p has unvisited adjacent vertices
               THEN  BEGIN
                      Let q be the first unvisited vertex
                      from the vertices adjacent to vertex p.
                      Walk along the edge (p,q), visit the vertex
                      bus q and push it onto the stack S
                  END
              ELSE   Remove node p from the stack S
        END
  END
```

As a result of the algorithm's operation, the traversed edges of the graph form one or more trees together with the visited vertices. If we assign an orientation to the traversed edges in accordance with the direction in which they are traversed during the execution of the algorithm, then we will obtain a set of rooted trees, and their roots will be those vertices that were placed in an empty stack during the operation of the algorithm.

---

(1) Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.
(2) Papadimitriou H, Steinglitz K. Combinatorial optimization. - M.: Mir, 1985. - 512 s.
(3) Lipsky V. Combinatorics for Programmers: Trans. with Polish. - M.: Mir, 1988. - 213 s.
(4) Kasyanov V.N., Sabelfeld V.K. Collection of tasks on the workshop on the computer. - M.: Nauka, 1986. - 272 s.

In the next step we will look ***at breadth-first search***.

Step 84.
Breadth-first search

In this step we will look at ***the breadth-first search algorithm***.

Let us now move on to another algorithm for searching on a graph, known as ***breadth-first search***. In breadth-first search, all neighbors of the current vertex are first looked at and only then does the graph move forward. Thus, the search is conducted in all possible directions simultaneously [1, p.131]:

```
(DEFUN BREADTHFIRST (LAMBDA (GRAPH ROOT)
 ; GRAPH - graph in the form of an adjacency structure ;
; ROOT - graph vertex from which the search begins ;
; Result: list of graph vertices in the order of traversal ;
   (BRFI
      GRAPH
      (LIST ROOT)
      (NEIGHBOUR3 ROOT GRAPH)
   )
))
; ------------------------------------ ;
(DEFUN BRFI (LAMBDA (GRAPH VISITED QUEUE)
; VISITED - a list of already viewed vertices ;
; QUEUE - a queue of vertices waiting to be viewed ;
   (COND ( (NULL QUEUE) (REVERSE VISITED) )
         ( T (COND ( (MEMBER (CAR QUEUE) VISITED)
                             (BRFI
                                GRAPH
                                VISITED
                                (CDR QUEUE)) )
                        ( T (BRFI
                                 GRAPH
                                 (CONS (CAR QUEUE) VISITED)
                                 (APPEND
                                    QUEUE
                                    (NEIGHBOUR3 (CAR QUEUE)
                                                GRAPH)) )) )
   )
))
; ------------------------------ ;
(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
   (COND ( (NULL (ASSOC X GRAPH)) NIL )
         (  T  (CDR (ASSOC X GRAPH)) )
   )
))
```

Test examples:

```
$ (SETQ G '((1 . (2 3 4))(2 . (1 3))(3 . (1 2 4))(4 . (1 3 5))(5 . (4))))
((1 . (2 3 4))(2 . (1 3))(3 . (1 2 4))(4 . (1 3 5))(5 . (4)))
$ (BREADTHFIRST G 4)
(4 1 3 5 2)
$ (BREADTHFIRST '((1 . (2 3 4)) (2 . (3)) (3 . (4))) 1)
(1 2 3 4)
$ (BREADTHFIRST '((1 . (2 3 4)) (2 . (3)) (3 . (4))) 2)
(2 3 4)
$ (BREADTHFIRST '((1 . (2 3 4)) (2 . (3)) (3 . (4))) 3)
(3 4)
$ (BREADTHFIRST '((1 . (2 3 4)) (2 . (3 1)) (3 . (4))) 2)
(2 3 1 4)
$ (BREADTHFIRST '((1 . (2 3)) (2 . (4 5)) (3 . (6 7))) 1)
(1 2 3 4 5 6 7)
```

The last test example is illustrated by the following graph:

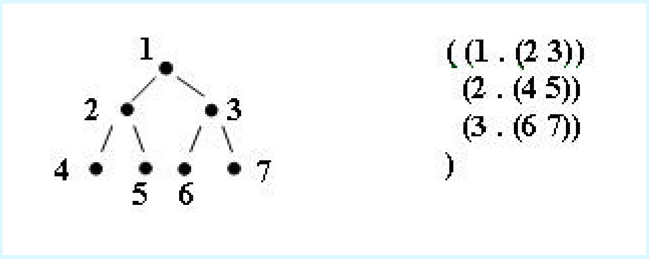


Fig. 1. Example of a graph

We see that in ***breadth-first search,*** the nearest neighbors of the last visited vertex are added to the end of the **QUEUE** list, while new vertices for viewing are taken from the beginning of this list, i.e. the **QUEUE** parameter is ***a queue,*** unlike the **PATH parameter of the DEFI** function, which works as a stack. This leads to the fact that during the operation of the **BRFI** function, the visited vertices in the **VISITED** list are arranged in non-increasing order of their distance from the starting vertex **ROOT**.

Notes.

1. *The algorithm for breadth-first search on a graph is described, for example, in the monographs [2, pp. 198-205], [3, pp. 88-91].*
2. ***The breadth-first tree traversal** method, sometimes called the horizontal order traversal method, is based on visiting the tree nodes from left to right, level by level down from the root.*

*The following **non-recursive** algorithm implements a **breadth-first** tree traversal using two queues **O1** and **O2** [4, pp.124-125]:*

```
Take empty queues O1 and O2.
Place the root in queue O1.
 WHILE   One of queues O1 and O2 is not empty DO
   IF  O 1 is not empty
         THEN  BEGIN
               Let p be the node at the head
               of queue O1.
               Visit vertex p and remove it from O1.
               Place all sons of vertex p in queue
                O2, starting with the eldest son
              END
        ELSE Take non-empty queue O2 as O1, and take the empty queue O1
              as O2.
```

[1] Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.
[2] Papadimitriou H, Steinglitz K. Combinatorial optimization. - M.: Mir, 1985. - 512 s.
[3] Lipsky V. Combinatorics for Programmers: Trans. with Polish. - M.: Mir, 1988. - 213 s.
[4] Kasyanov V.N., Sabelfeld V.K. Collection of tasks on the workshop on the computer. - M.: Nauka, 1986. - 272 s.

From the next step we will start looking at ***examples of graph algorithms***.

Step 85.
Calculate connectivity components

In this step we will consider ***the algorithm for calculating the connectivity components***.

Graph search can be used to answer some questions about the structure of a graph. We will consider two search-related problems here:

- ***calculation of the connectivity components of a graph*** and
- ***construction of a spanning tree of a connected graph.***

To calculate ***the connectivity components*** of the graph, we will use the already written **DEPTHFIRST** function . Let us consider the operation of the **DEFI** function once again . Note that when a new vertex is added to the **VISITED** list of viewed vertices, the **PATH** list is the path from this vertex to the initial one. Therefore, at the end of the operation, the **VISITED** list contains only those vertices that can be connected to the initial one. It can be shown [2] that this list contains all vertices that have this property.

Thus, we can say that the **DEPTHFIRST** function computes the list of vertices of the connected component of the **ROOT** vertex, and now we are ready to solve the problem of constructing all the connected components of the given graph.

The easiest way to do this is in two stages.

1. First, we construct lists of vertices of connectivity components. Program [1, pp.129-130].

```
(DEFUN CONNLISTS (LAMBDA (GRAPH)
 ; GRAPH - a graph in the form of an adjacency structure ;
 ; Result: a list of lists of vertices of connectivity components ;
   (CONNLSTS GRAPH NIL)
))
; --------------------------------- ;
(DEFUN CONNLSTS (LAMBDA (GRAPH LISTS)
   (COND ( (NULL GRAPH) LISTS )
         ( T (COND ( (NULL (LMEMBER (CAAR GRAPH) LISTS))
                      (CONNLSTS
                          (CDR GRAPH)
                          (CONS (DEPTHFIRST
                                    GRAPH
                                    (CAAR GRAPH))
                                LISTS)
                      )
                   )
                   (  T  (CONNLSTS (CDR GRAPH) LISTS) )
             )
         )
   )
))
; --------------------------------- ;
(DEFUN LMEMBER (LAMBDA (VERTEX LISTS)
; Checks if the vertex VERTEX is contained in any of the ;
; lists that make up LISTS ;
   (AND
       LISTS
       (OR (MEMBER  VERTEX (CAR LISTS))
           (LMEMBER VERTEX (CDR LISTS))
       )
   )
))
; ---------------------------------- ;
(DEFUN DEPTHFIRST (LAMBDA (GRAPH ROOT)
; GRAPH - a graph in the form of an adjacency structure ;
; ROOT - the graph vertex from which the search begins ;
; Result: a list of graph vertices in the order of traversal ;
   (COND ( (NULL GRAPH) NIL )
         (  T  (DEFI GRAPH (LIST ROOT) (LIST ROOT)) )
   )
))
; ------------------------------------ ;
(DEFUN DEFI (LAMBDA (GRAPH VISITED PATH)
; VISITED - a list of already viewed vertices ;
; PATH - a list of vertices defining the viewing path ;
   (COND ( (NULL PATH) (REVERSE VISITED) )
         ( T  (COND ( (NULL (EXPND GRAPH VISITED (CAR PATH)))
                       (DEFI GRAPH
                             VISITED
                             (CDR PATH))
                    )
                    (  T  (DEFI GRAPH
                                (CONS (EXPND GRAPH VISITED
                                             (CAR PATH))
                                      VISITED)
                                (CONS (EXPND GRAPH VISITED
                                             (CAR PATH))
                                      PATH))
                    )
              )
         )
   )
))
; ---------------------------------------- ;
(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)
; Select the next unvisited vertex, ;
; adjacent to VERTEX ;
   (COND ( (NULL (NEIGHBOUR3 VERTEX GRAPH)) NIL )
         (  T  (FIRSTNOTVISITED
                          VISITED
                          (NEIGHBOUR3 VERTEX GRAPH))
         )
   )
))
; ------------------------------------------- ;
(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)
   (COND ( (NULL VLIST) NIL )
         (  T  (COND ( (NULL (MEMBER (CAR VLIST) VISITED))
                          (CAR VLIST) )
                     (  T  (FIRSTNOTVISITED
                                        VISITED
                                        (CDR VLIST)) )
               )
         )
   )
))
; -------------------------------- ;
(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
   (COND ( (NULL (ASSOC X GRAPH)) NIL )
         (  T  (CDR (ASSOC X GRAPH)) )
   )
))
```

Test examples:

```
$ (CONNLISTS '((1 . (2 3))   (2 . (3)) (5 . ())))
((5) (1 2 3))
$ (CONNLISTS '((1 . (2 3))   (2 . (3)) (5 . ()) (6 . ()))
((6) (5) (1 2 3))
$ (CONNLISTS '((1 . (2 3))   (2 . (3)) (5 . (6))))
((5 6) (1 2 3))
$ (CONNLISTS '((1 . (2 3 5)) (2 . (3)) (5 . (6))))
((1 2 3 5 6))
```

2. Now it is easy to write a function that calculates the connectivity components themselves, represented as adjacency structures. To do this, we will use the following auxiliary functions that allow us to extract connectivity components:

```
(DEFUN F2 (LAMBDA (GRAPH LST)
   (COND ( (NULL LST) NIL )
         (  T  (CONS (F1 GRAPH (CAR LST))
                     (F2 GRAPH (CDR LST))) )
   )
))
```

```
; ------------------------- ;
(DEFUN F1 (LAMBDA (GRAPH LST)
   (COND ( (NULL GRAPH) NIL )
         ( (MEMBER (CAAR GRAPH) LST)
              (CONS (CAR GRAPH) (F1 (CDR GRAPH) LST)) )
         (  T  (F1 (CDR GRAPH) LST) )
   )
))
```

Test example:

```
$ (F2 '((1 . (2 3)) (4 . (5 6)) (5 . (4)) (3 . (2 1))) '((1 2 3) (4 5 6)))
(((1 2 3) (3 2 1) ((4 5 6) (5 4)))
```

---

[1] Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.
[2] Lipsky V. Combinatorics for Programmers: Trans. with Polish. - M.: Mir, 1988. - 213 s.

In the next step we will consider ***the construction of a spanning tree of a connected graph***.

Step 86.
Building a spanning tree of the connected graph

In this step we will consider ***the algorithm for constructing a spanning tree***.

Let us now turn to the problem of constructing ***a spanning tree of a connected graph*** . To do this, let us recall that in depth-first search, a new vertex is added to the list of already traversed vertices only if it is not in this list. Therefore, the sequence of vertices corresponding to the extension of the **PATH** list when the **DEFI** function is running is ***a simple (i.e., without self-intersections) path in the graph being viewed***.

Following these remarks, one can write a function whose value is a list of edges representing the spanning tree of the graph. Program [1, pp. 130-131].

```
(DEFUN SPANDF (LAMBDA (GRAPH ROOT)
 ; GRAPH - graph in the form of adjacency structure ;
; ROOT - root of spanning tree ;
; Result: list of edges of spanning tree obtained ;
; using depth-first search ;
   (COND ( (NULL GRAPH) NIL )
         (  T  (SPDF GRAPH (LIST ROOT) (LIST ROOT)) )
   )
))
; ------------------------------------ ;
(DEFUN SPDF (LAMBDA (GRAPH VISITED PATH)
; VISITED - a list of already viewed vertices ;
; PATH - a list of vertices defining the viewing path ;
   (COND ( (NULL PATH) NIL )
         (  T  (COND ( (NULL (EXPND GRAPH VISITED
                                    (CAR PATH)))
                          (SPDF GRAPH VISITED (CDR PATH))
                     )
                     (  T  (CONS
                               (CONS (CAR PATH)
                                     (EXPND GRAPH VISITED
                                            (CAR PATH)))
                               (SPDF
                                   GRAPH
                                   (CONS (EXPND GRAPH
                                                VISITED
                                                (CAR PATH))
                                         VISITED)
                                   (CONS (EXPND GRAPH
                                                VISITED
                                                (CAR PATH))
                                         PATH))) )) )
   )
))
; ---------------------------------------- ;
(DEFUN EXPND (LAMBDA (GRAPH VISITED VERTEX)
; Select the next unvisited vertex ;
; adjacent to VERTEX ;
   (COND ( (NULL (NEIGHBOUR3 VERTEX GRAPH)) NIL )
         (  T  (FIRSTNOTVISITED
                             VISITED
                             (NEIGHBOUR3 VERTEX GRAPH))
         )
   )
))
; ------------------------------------------- ;
(DEFUN FIRSTNOTVISITED (LAMBDA (VISITED VLIST)
   (COND ( (NULL VLIST) NIL )
         (  T  (COND ( (NULL (MEMBER (CAR VLIST) VISITED))
                          (CAR VLIST) )
                     (  T  (FIRSTNOTVISITED
                                         VISITED
                                         (CDR VLIST)) )) )
   )
))
; -------------------------------- ;
(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
   (COND ( (NULL (ASSOC X GRAPH)) NIL )
         (  T  (CDR (ASSOC X GRAPH)) )
   )
))
```

Test example:

```
   $ (SPANDF '((1 . (2 3)) (2 . (4 5)) (3 . (6 7))) 1)
   ((1 . 2) (2 . 4) (2 . 5) (1 . 3) (3 . 6) (3 . 7))
```

Now we ask ourselves what properties will a spanning tree constructed using the ***breadth-first search*** procedure have? We have already mentioned that in breadth-first search the scanned vertices are arranged in order of non-decreasing distance from the initial vertex. Therefore, in a spanning tree constructed using breadth-first search, the only path connecting any vertex to the initial one coincides with the shortest path in the original graph between these two vertices.

The changes in the BRFI function required to construct a spanning tree using breadth-first search are more significant than those in the **DEFI** function. The point is that in breadth-first search we do not need information about the predecessor of the node in question - in this sense, breadth-first search is non-local. Therefore, instead of a queue of nodes to be scanned, we introduce ***a queue*** consisting directly of the elements of the adjacency structure.

Program [1, p.132-133].

```
(DEFUN SPANBF (LAMBDA (GRAPH ROOT)
 ; GRAPH - graph in the form of an adjacency structure ;
; ROOT - the root of the spanning tree ;
```

```
  ; Result: spanning tree ;
     (SPBF GRAPH (LIST ROOT) (CDR (ASSOC ROOT GRAPH))
           NIL (CAR (ASSOC ROOT GRAPH)))
  ))
  ; ------------------------------------------------- ;
  (DEFUN SPBF (LAMBDA (GRAPH VISITED HEAD QUEUE FATHER)
  ; QUEUE - queue of vertex lists to view ;
  ; HEAD - list of vertices to view ;
  ; FATHER - ancestor of the vertices being viewed ;
     (COND ( (NULL HEAD)
                (COND ( (NULL QUEUE) NIL )
                      (  T  (SPBF GRAPH VISITED
                                  (CDAR QUEUE) (CDR QUEUE)
                                  (THAT FRIEND)) )
                )
           )
           ( T  (COND ( (MEMBER (CAR HEAD) VISITED)
                           (SPBF GRAPH VISITED
                                 (CDR HEAD) QUEUE
                                 FATHER) )
                      (  T  (CONS
                                (CONS FATHER (CAR HEAD))
                                (SPBF
                                   GRAPH
                                   (CONS (CAR HEAD) VISITED)
                                   (CDR HEAD)
                                   (APPEND
                                      QUEUE
                                      (LIST (ASSOC (CAR HEAD)
                                                   GRAPH)))
                                   FATHER)) )) )
     )
  ))
```

Test examples:

```
  $ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 1)
  ((1 . 2) (1 . 4) (2 . 3))
  $ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 2)
  ((2 . 1) (2 . 3) (1 . 4))
  $ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 3)
  ((3 . 1) (3 . 4) (1 . 2))
  $ (SPANBF '((1 . (2 4)) (2 . (1 3)) (4 . (1 3)) (3 . (1 4))) 4)
  ((4 . 1) (4 . 3) (1 . 2))
  $ (SPANBF '((1 . (2 3)) (2 . (4 5)) (3 . (6 7))) 1)
  ((1 . 2) (1 . 3) (2 . 4) (2 . 5) (3 . 6) (3 . 7))
```

Note that when the **SPBF function is running, the HEAD** parameter is always a list of neighbors of the vertex specified by the **FATHER** parameter, which allows one to know the predecessor of the current vertex.

---

(1) Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.

In the next step we will get acquainted with ***algorithms with backtracking***.

Step 87.
Algorithms with backtracking

In this step we will look at ***the implementation and use of backtracking algorithms***.

So far, we have only considered graph algorithms in which we never had to go ***back*** during their execution.

If, for example, a vertex was viewed during a search on a graph, it will remain in the list of viewed vertices until the end of the algorithm.

However, there is a large class of problems for which ***such algorithms are unknown***. Perhaps the best known of these is ***the traveling salesman problem*** (see, for example, [2]) - the problem of finding the shortest route between **N** cities, where only some of the cities are directly connected by roads.

Here we consider the problem of finding a route that starts and ends in the same city and, in addition, does not visit the same city twice. This problem is known as ***the problem of finding a Hamiltonian cycle in a graph*** . As is clear from the above, ***a Hamiltonian cycle*** is a simple closed path containing all the vertices of the graph. Note that not every graph has a Hamiltonian cycle.

The algorithm we will use to find Hamiltonian cycles is called "search with return". It is based on the concept of ***a partial solution***. Let the solution to the problem be represented as a certain sequence. In our case, this is simply a sequence of all the vertices of the graph that satisfies obvious constraints. The initial segment of the sequence that satisfies the constraints that determine the complete solution is called ***a partial solution***.

A partial solution to a Hamiltonian cycle is any sequence of vertices that defines a simple path.

If a new element is added to a sequence representing a partial solution so that the extended sequence is again a partial solution, then the new sequence is called ***a continuation of the partial solution***. A continuation is usually ambiguous.

The idea of a search with return is to start with a trivial partial solution and continue it sequentially until either a complete solution is obtained or the continuation becomes impossible. In the latter case, we go back a step and try to construct another continuation. If the search space is finite, then either we obtain a complete solution or we are convinced that it is impossible to construct one, since a complete enumeration of possible continuations has been performed.

After these remarks, we present ***an implementation of the algorithm for finding a Hamiltonian path in a connected undirected graph***. Program [1, pp. 134-135].

```
(DEFUN HAMILTCYCLE (LAMBDA (GRAPH)
 ; GRAPH - graph as adjacency structure ;
; Result: Hamiltonian cycle as list of vertices, ;
; NIL - if Hamiltonian cycle does not exist ;
   (COND ( (NULL GRAPH) NIL )
         (  T  (COND ( (NULL (CDR GRAPH))
                              (LIST (CAAR GRAPH)))
                     ( T (HC GRAPH (CAAR GRAPH)
                                   (LIST (CAR GRAPH))
                                   (CDAR GRAPH)
                              )
                     )
               )
         )
   )
))
; ---------------------------------------- ;
(DEFUN HC (LAMBDA (GRAPH START VISITED SONS)
; START - the first vertex of the graph ;
; VISITED - a list of visited vertices ;
; SONS - neighbors of the viewed vertex ;
   (COND ( (NULL SONS) NIL )
         (  T  (COND ( (AND (MEMBER START SONS)
                            (EQ (LENGTH GRAPH)
                                (LENGTH VISITED))
                       )
                          (REVERSE VISITED)
                     )
                     ( T (COND
                              ( (MEMBER (CAR SONS) VISITED)
                                   (HC GRAPH START VISITED
                                            (CDR SONS)) )
                              (  T  (OR (HC GRAPH START
                                          (CONS
                                            (CAR SONS)
                                            VISITED)
                                          (NEIGHBOUR3
                                            (CAR SONS)
                                            GRAPH)
                                       )
                                       (HC
                                          GRAPH START VISITED
                                          (CDR SONS))) )) )
               )
         )
   )
))
; ------------------------------- ;
(DEFUN NEIGHBOUR3 (LAMBDA (X GRAPH)
   (COND ( (NULL (ASSOC X GRAPH)) NIL )
         (  T  (CDR (ASSOC X GRAPH)) )
   )
))
```

Test examples:

```
   $ (HAMILTCYCLE '((1 . (2 6)) (2 . (1 3 4)) (3 . (2 4))
   (4 . (2 3 5)) (5 . (4 6)) (6 . (1 5))))
   (1 2 3 4 5 6)
   $ (HAMILTCYCLE '((A . (F D C)) (F . (E D A B))
   (E . (D F B)) (D . (E F A)) (C . (A B)) (B . (F E C))))
   (A F D E B C)
```

Let us analyze the operation of the **HAMILTCYCLE** function. For an empty graph, its value is equal to the empty list **NIL**, and for a graph consisting of a single vertex, it is equal to the list containing this vertex. If the graph consists of more than two vertices, then the HC function is called. It is this function that continues the partial solution. The partial solution is presented as a **VISITED** list - this is a list of viewed vertices in the reverse order of their viewing. The **SONS** list is a list of neighbors of the first vertex in the **VISITED** list. From the point of view of the partial solution, this vertex is just the last one. Therefore, vertices from the **SONS** list can be included in the continuations of the partial solution. If this list is empty, then the continuation of the partial solution is impossible and the **HC** function returns the value **NIL**. Otherwise, it is checked whether it is possible to close the Hamiltonian cycle at the next step. This is possible if, firstly, the initial vertex is contained in the **SONS** list and, secondly, if all vertices of the graph have been visited. Since each node is visited no more than once by the condition (this follows from the definition of a partial solution), it is sufficient to check that the lengths of **the GRAPH** and **VISITED** lists are equal. If both of the above conditions are met, then the **VISITED** list is the desired cycle.

Let us now turn to the case where continuation is possible, but does not immediately lead to the construction of a complete solution. Our goal is to continue a partial solution. To do this, we look through the **SONS** list until we encounter a vertex that is not in the **VISITED** list. If such a vertex is not found, then continuation is impossible. Otherwise, the expression is evaluated

```
   (OR
      (HC GRAPH START (CONS (CAR SONS) VISITED)
          (NEIGHBOUR3 (CAR SONS) GRAPH))
      (HC GRAPH START VISITED (CDR SONS))
   )
```

This is the key point of backtracking. If the first call to **HC** evaluates to a value other than **NIL**, then the continuation of **(CONS (CAR SONS) VISITED)** has been built up to a complete solution. This solution is returned to the top level. Otherwise, a second call to **HC** attempts to find a continuation other than **(CONS (CAR SONS) VISITED)**.

Note that the entire return mechanism is hidden in recursion. This makes the program clear and compact.

The HAMILTCYCLE function will terminate once it finds the first Hamiltonian cycle or looks through all possibilities. You can also set ***the task of finding all Hamiltonian cycles of a given graph***.

Note that the backtracking search algorithm can be interpreted as a search in a graph. The vertices of this graph are partial solutions, and the edges are those pairs of partial solutions, one of which is a direct continuation of the other. The choice of one or another search algorithm on the graph determines the strategy of backtracking search.

Notes .

1. *Don't be lazy and take a look at [3, pp.108-114]!*
2. *[1, p.137]. We have considered the implementation of several simple graph algorithms in **LISP**. These implementations are extremely simple and clear, which is achieved, on the one hand, by the choice of data representation, and on the other, by using purely functional programming methods. The functional approach has, in addition to obvious advantages, disadvantages. The latter primarily include the insufficient efficiency of the resulting programs. This concerns both performance and memory use.*

   The effectiveness of a program is often of primary importance. Let us therefore say a few words about how the effectiveness of the above programs could be increased.

   *One of the reasons for the relatively low performance of our programs is the overhead of organizing recursion. The program's performance could be increased by replacing recursion **with iteration**, which is always possible in principle [4]. However, this can lead to very complex and obscure programs that cause many difficulties when debugging.*

   *Another reason for inefficiency is the following: Our algorithms very often involve searching **for a given element in a list** .*

   The general search scheme looked like this:

```
   (DEFUN SUCHTHAT (LAMBDA (P LST)
    ; The SUCHTHAT function selects the first element from the list LST
   that has the P property. If
   there is no such element, then the NIL value is returned;
      (COND ( (NULL LST) LST )
            ( (P (CAR LST)) (CAR LST) )
            ( T (SUCHTHAT P (CDR LST)) )
      )
   ))
```

*Here **P** is some predicate that checks the search condition. Such a search is called **sequential** . The time spent on it is proportional, on average, to the length of the list being searched. If the search is performed many times, then it may make sense to organize the data differently. For example, you can use the so-called **binary search tree**.*

The speed of searching in such a tree, generally speaking, depends on the order in which nodes are included in it, but in many cases it can be proportional to the binary logarithm of the total number of nodes.

Search tree search algorithm:

```
(DEFUN SEARCH (LAMBDA (A TREE)
 ; The SEARCH function searches for element A in the TREE. ;
; If successful, the function returns the subtree of
the TREE ; ; in which element A is the root; if
the search is unsuccessful, the function returns NIL ;
   (COND ( (NULL TREE)          NIL  )
         ( (EQUAL A (CAR TREE)) TREE )
         ( (LESSP A (CAR TREE)) (SEARCH A (CADR  TREE)) )
         (         T            (SEARCH A (CADDR TREE)) )
   )
))
```

---

(1) Kryukov A.P., Radionov A.Ya., Taranov A.Yu., Shablygin E.M. Programming in R-Lisp. - M.: Radio and communication, 1991. - 192 p.
(2) Reingold E, Nievergelt Y, Deo N. Combinatorial algorithms. Theory and practice. - M.: Mir, 1980. - 476 s.
(3) Lipsky V. Combinatorics for Programmers: Trans. with Polish. - M.: Mir, 1988. - 213 s.
(4) Barron D. Recursive methods in programming. - M.: Mir, 1974. - 80 s.

In the next step we will begin to get acquainted with the pragmatics of the **LISP *language*** .

Step 88.
LISP Pragmatics. General Provisions

In this step we will look at ***the concept of pragmatics***.

In relation to sign systems, in particular programming languages, we speak of their ***syntax*** - the rules for forming messages, ***semantics*** - the rules for interpreting a message by the person to whom it is addressed, and ***pragmatics***, which compares messages with the goals and intentions of the person from whom they come.

Let's look into explanatory dictionaries [1-4].

Pragmatics (from the Greek **pragma, pragmatos - *business, action***) -

1. The relationship of symbols and groups of symbols to their interpretation and use.
2. A section of semiotics that studies the attitude of the user of a sign system (for example, a programming language) to the sign system itself; pragmatics determines how the perceiving system, based on information not embedded in the perceived expression, selects from a variety of interpretations the most appropriate for a given case. In computer science, the evaluation and comparison of various computer languages, programs, and systems according to the criteria of usefulness, advantage, and efficiency.

The pragmatics of a programming language is, in essence, a programming methodology, i.e. a description of methods and techniques that allow, based on the formulation of a problem, to create a program for its solution. Some semantically correct programs may turn out to be completely unacceptable pragmatically.

Some problems, although stated extremely simply, have no algorithm for solving them. Such is, for example, the problem of ***checking whether any given program is semantically correct***. Problems of this kind are called ***algorithmically undecidable***.

Because of the existence of algorithmically unsolvable problems, the subject of pragmatics becomes somewhat vague and undefined - in general, it is impossible to give any recommendations that would guarantee a solution from the formulation of a problem. In addition, because of the diversity of problems solved with the help of a computer, those recommendations that can be given are either too general in nature, or, on the contrary, too specific, and relate to a narrow class of problems. Programming theory, although it can boast of a number of significant achievements, is oriented more towards developers of algorithmic languages and computer software than towards users. Many of its recommendations are merely formulations of problems, the solution of which will require many more years of work by system programmers, if it is ever achieved at all. Therefore, an important role in mastering the skills of compiling programs is played by ***the analysis of examples of varying degrees of complexity from different classes of problems***.

In view of this situation with pragmatics, more attention in the description of programming languages is paid to their syntax and semantics [5, pp. 4-7; 6, pp. 76-77].

With this in mind, in the following steps we will look at the application of the **LISP** language to solving a variety of programming problems.

---

(1) Borkovsky A.B. English-Russian Dictionary of Programming and Computer Science (with Explanations). - M.: Russian Language, 1989. - 335 p.
(2) Zamorin A.P., Markov A.S. Explanatory Dictionary of Computer Science and Programming: Basic Terms. - M.: Russian Language, 1988. - 221 p.
(3) Explanatory Dictionary of Computing Systems / Ed. by V. Illingworth, E.L. Glazer, I.K. Pyle. - M.: Mashinostroenie, 1989. - 568 p.
(4) Pershikov V.I., Savinkov V.M. Explanatory Dictionary of Computer Science. - M.: Finance and Statistics, 1991. - 543 p.
(5) Lavrov S.S. Basic Concepts of Programming Languages. - M.: Finance and Statistics, 1982. - 80 p.
(6) Bauer F.L., Goos G. Computer Science. Introductory Course: In 2 parts. Part 1. translated from German. - M.: Mir, 1990. - 336 p.; Part 2. translated from German. - M.: Mir, 1990. - 423 p.

From the next step we will start looking at ***examples of programs in the*** **LISP** language.

Step 89.
Pattern matching

In this step, we will look at ***the implementation of the pattern matching task***.

Often, when processing symbolic information, it is necessary to solve the question of the equivalence of two symbolic objects. Such a task usually arises when some rules are specified in the form of a left part (pattern), a right part, and a condition. If the symbolic information being processed can be identified with the pattern in such a way that the condition is met, then the right part, which is part of the given rule, is used for its further processing.

Pattern matching is the process of comparing symbolic expressions. It is an integral part of any advanced computer-based analytical computing system.

Unlike, for example, ***Refal*** **[1, 2], LISP** does not have a built-in mechanism for working with patterns. However, a program that allows matching fairly complex patterns can be easily written. In this sense, **LISP** is said to be a good language for implementing pattern matching.

So suppose we have two lists of atoms: ***a sample*** and ***a fact*** . Often the list corresponding to a fact will be used to represent true statements about some real or imaginary world.

Below we will construct a **MATCH** function that will match one sample and one fact [3, p.380-384].

When a sample that does not contain special atoms (**?** and *****) is compared with some fact, the comparison is considered successful only if they are the same, i.e. each position is occupied by the same atoms.

We will follow the strategy of moving along the pattern and along the fact atom by atom, checking whether the pattern atom and the fact atom match at each position.

```
(DEFUN MATCH (LAMBDA (PD)
 ; P - function; D - factor ;
   (COND ( (AND (NULL P) (NULL D)) T  )
         ( (OR (NULL P) (NULL D)) NIL )
         ( (EQ (CAR P) (CAR D)) (MATCH (CDR P) (CDR D)) )
   )
))
```

Let's extend the useful capabilities of the **MATCH** function . Let's assume that the sample contains a special atom **?** . The selected atom **?** has the property that it is comparable to any atom.

Let's generalize the MATCH function:

```
(DEFUN MATCH (LAMBDA (PD)
 ; P - function; D - factor ;
   (COND ( (AND (NULL P) (NULL D)) T  )
         ( (OR (NULL P) (NULL D)) NIL )
         ( (OR (EQ (CAR P) '?) (EQ (CAR P) (CAR D)))
               (MATCH (CDR P) (CDR D))
         )
   )
))
```

In **MATCH,** recursion occurs while the current first atom in the pattern is either **?** or matches the current first atom in the fact.

The * atom also extends the power of the **MATCH** function by being matchable to one or more atoms. Patterns containing the ***** atom may be matchable to facts that have more atoms than the pattern.

Let's continue generalizing the **MATCH** function:

```
(DEFUN MATCH (LAMBDA (PD)
 ; P - function; D - factor ;
   (COND ( (AND (NULL P) (NULL D)) T  )
         ( (OR (NULL P) (NULL D)) NIL )
         ( (OR (EQ (CAR P) '?) (EQ (CAR P) (CAR D)))
               (MATCH (CDR P) (CDR D))
         )
         ( (EQ (CAR P) '*)
               (COND ( (MATCH (CDR P) D) )
                     ( (MATCH (CDR P) (CDR D)) )
```

```
                         ( (MATCH P (CDR D)) )
                    )
               )
          )
     ))
```

Test examples:

```
   $ (MATCH '(* is *) '(My Name is John))
   T
   $ (MATCH '(Color is ?) '(Color is Red))
   T
```

The asterisk-toggled COND clause provides a recursive call to **MATCH** . One of three possibilities occurs:

1) * is not comparable to anything, in this case it is discarded and the rest of the sample is compared;

2) * is comparable to one atom, in this case both * and this atom are discarded;

3) * is matched to two or more atoms, in which case we proceed in a recursive manner, keeping * in the pattern, discarding the first atoms to which it matches successfully.

The second term in the above COND clause is actually redundant if * can match any number of atoms, including zero. It can be eliminated, and **MATCH** will still work. In that case, **MATCH** strips off **the** * and matches the rest of **P** against the untouched list **D**, or else keeps **the** * and moves one atom into the data, assuming that it is one of the atoms that matches *.

On the other hand, in this **COND** function the first clause may be omitted, but not the second. Then * must necessarily be comparable to at least one atom, as originally defined [3, p.383]. The next direction of improvement lies in generalizing the **MATCH** function so that it assigns specific values when the match is successful [3, pp.384-387].

Let atoms whose P-names begin with > or * act like **?** and * in the matching process, but in case of success their values are the things they match. Clearly, atoms whose P-names begin with > should have atoms as values, while atoms whose P-names begin with * should have lists of matched atoms as values. The notation > used here is intended to suggest "pushing" values into atoms whose P-names contain the "preposition" >.

For example:

```
   $ (MATCH '(PLUS >A >B) '(PLUS 2 3))
   T
   $ A
   2
   $ B
   3
   $ (MATCH '(*L is *R) '(My Name is John))
   T
   $ L
   (My Name)
   $ R
   (John)
```

Matching with > and assignment can be implemented using the **ATOMCAR** and **ATOMCDR** functions:

```
    (DEFUN ATOMCAR (LAMBDA (X)
     ; Select the first character of the name of a literal atom ;
       (CAR (UNPACK X))
   ))
```

```
; ---------------------- ;
(DEFUN ATOMCDR (LAMBDA (X)
% Selecting the "tail" of a literal atom name %
   (PACK (CDR (UNPACK X)))
))
```

Note the use of **SET** to assign the value of the sample variable to the item in the data that it matches:

```
( (AND (EQ (ATOMCAR P) >) (MATCH (CDR P) (CDR D)))
         (SET (ATOMCDR P) (CAR D))
)
```

Variables with * are handled in the same way, this time by assigning them values as lists of atoms that map to those variables, using the following:

```
( (EQ (ATOMCAR (CAR P)) '*)
   (COND ( (MATCH (CDR P) (CDR D))
            (SET (ATOMCDR (CAR P)) (LIST (CAR D))) T)
         ( (MATCH P (CDR D))
            (SET (ATOMCDR (CAR P))
              (CONS (CAR D) (EVAL (ATOMCDR (CAR P))))) T)
   )
)
```

Note that no assignment and no application **of SET** occurs until the entire recursion has been performed and the match is known to have succeeded.

Here is the final version of the **MATCH** function:

```
(DEFUN MATCH (LAMBDA (PD)
 ; P - function; D - factor ;
   (COND ( (AND (NULL P) (NULL D))   T  )
         ( (OR  (NULL P) (NULL D)) NIL )
         ( (OR (EQ (CAR P) '?) (EQ (CAR P) (CAR D)))
              (MATCH (CDR P) (CDR D)) )
         ( (AND (ATOM (CAR P)) (EQ (ATOMCAR (CAR P)) '>)
               (MATCH (CDR P) (CDR D)))
               (SET (ATOMCDR (CAR P)) (CAR D)) T )
         ( (EQ (CAR P) '*)
              (COND ( (MATCH (CDR P) (CDR D)) )
                    ( (MATCH P (CDR D))) ) )
         ( (AND (ATOM (CAR P)) (EQ (ATOMCAR (CAR P)) '*))
            (COND ( (MATCH (CDR P) (CDR D))
                    (SET (ATOMCDR (CAR P)) (LIST (CAR D))) T)
                  ( (MATCH P (CDR D))
                    (SET (ATOMCDR (CAR P))
                      (CONS (CAR D) (EVAL (ATOMCDR (CAR P)))))
                  ) T ) ) )
   )
))
; ---------------------- ;
(DEFUN ATOMCAR (LAMBDA (X)
; Select the first character of the name of a literal atom ;
   (CAR (UNPACK X))
))
; ---------------------- ;
(DEFUN ATOMCDR (LAMBDA (X)
; Selecting the "tail" of the name of a literal atom;
   (PACK (CDR (UNPACK X)))
))
```

The matching system described is simple because it is focused on matching one list of atoms to another. Moreover, there is no notion of closeness: either the matching is successful or it is not. Working with networks, frames, or other large bodies of knowledge can be much more difficult because the following possibilities arise:

- the matching system must produce some score on a scale ranging from no match at all to a perfect match;
- the matching system must work with "more" diverse structures.

Enabling these features can be an extremely difficult task.

---

(1) Basic REFAL and its implementation on computers. / TsNIPIASS. - M., 1977. - 238 p.
(2) Turchin V.F. Basic REFAL. Description of the language and basic programming techniques. - M.: TsNIPIASS, 1974. - 258 p.
(3) Winston P. Artificial Intelligence. - M.: Mir, 1980, pp.303-512.

In the next step we ***will model a psychiatrist***.

Step 90.
Simulated Psychiatrist

In this step we will look at ***using the material we have learned to model human activity***.

With what we have at present, it is not difficult to write a simple version of a program that, when interacting with a person sitting at a terminal, resembles a certain type of psychiatrist talking to a patient. This program is a loop using **COND** functions containing key words and phrases along with the appropriate response to them [1].

Communication with the user occurs using the **READ** and **PRINT** functions. Note also that if the word **YEAR** is encountered, the **FLAG** atom receives the current value of **T** . Then later, if nothing else worked, the program's reaction is

```
                    (YOU WERE TALKING ABOUT...)
```

seems quite reasonable.

The proposed program could be and has been developed using very sophisticated scenarios, but even in our trivial version it is capable of a short dialogue.

```
   (DEFUN DOCTOR (LAMBDA NIL
    ; Simulation of a psychiatrist ;
      (PRINT 'SPEAK!)
      (SETQ FLAG NIL)
      (LOOP
         (SETQ S (READ))
         (COND ( (MATCH '(I CARE ABOUT *L) S)
                   (PRINT (APPEND '(HOW LONG HAVE YOU BEEN WORRIED)
                                   L)) )
               ( (MATCH '(* ЛЕТ) S)
                   (SETQ FLAG T)
                   (PRINT (APPEND '(TELL US MORE ABOUT)
                                   L)) )
               ( (MATCH '(* COMPUTERS *) S)
                   (PRINT (APPEND '(COMPUTERS ARE SCARY)
                                   L)) )
               ( (OR (MATCH '(NO) S) (MATCH '(YES) S))
                  (PRINT '(PLEASE DON'T BE BRIEF))
              ( MOTHER
                  (SETQ FLAG NIL)
                  (PRINT (LIST '(YOU TALKED ABOUT) L)) )
              ( T (PRINT '(LET'S CONTINUE OUR DIALOGUE!)) )
        )
     )
  ))
  ; ---------------------- ;
  (DEFUN MATCH (LAMBDA (P D)
     (COND   ( (AND (NULL P) (NULL D))  T  )
             ( (OR  (NULL P) (NULL D)) NIL )
             ( (OR (EQ (CAR P) '?) (EQ (CAR P) (CAR D)))
                 (MATCH (CDR P) (CDR D)) )
             ( (AND (ATOM (CAR P)) (EQ (ATOMCAR (CAR P)) '>)
                  (MATCH (CDR P) (CDR D)))
                 (SET (ATOMCDR (CAR P)) (CAR D)) T )
             ( (EQ (CAR P) '*)
                 (COND ( (MATCH (CDR P) (CDR D)) )
                       ( (MATCH P (CDR D))) ) )
             ( (AND (ATOM (CAR P)) (EQ (ATOMCAR (CAR P)) '*))
                 (COND ( (MATCH (CDR P) (CDR D))
                          (SET (ATOMCDR (CAR P))
                               (LIST (CAR D))) T)
                       ( (MATCH P (CDR D))
                          (SET (ATOMCDR (CAR P))
                               (CONS (CAR D)
                                   (EVAL (ATOMCDR (CAR P))))
                          ) T ) ) )
     )
  ))
  ; ---------------------- ;
  (DEFUN ATOMCAR (LAMBDA (X)
  ; Select the first character of the name of a literal atom ;
     (CAR (UNPACK X))
  ))
  ; ---------------------- ;
  (DEFUN ATOMCDR (LAMBDA (X)
  ; Selecting the "tail" of the name of a literal atom;
     (PACK (CDR (UNPACK X)))
  ))
  ; ----------------------------- ;
  (DEFUN APPEND (LAMBDA (LST1 LST2)
  ; The APPEND function returns a list consisting of ;
  ; the elements of LST1 appended to LST2 ;
     (COND ( (NULL LST1) LST2 )
           ( (NULL LST2) LST1 )
           (  T  (CONS (CAR LST1)
                       (APPEND (CDR LST1) LST2)) )
     )
  ))
```

Test example:

```
(SPEAK!)
(I'M WORRIED ABOUT MOTYA)
(HOW LONG HAVE YOU BEEN WORRIED ABOUT MOTYA)
(5 YEARS)
(TELL ME MORE ABOUT MOTYA)
(HE AND COMPUTERS ARE GREAT FRIENDS!)
(COMPUTERS SCARE MOTYA)
(NO)
(PLEASE DON'T BE BRIEF)
(SORRY)
(YOU WERE TALKING ABOUT MOTYA)
```

```
    (YES)
    (PLEASE DON'T BE BRIEF)
    ...
```

It is important to note that the **DOCTOR** program does not actually understand the person at the terminal. It does not build a model of the problems being discussed, the program relies entirely on superficially catching keywords.

Although the **DOCTOR** function should be considered purely as a toy, it nevertheless shows how an effective pattern matching function can simplify programming. The interesting thing about the **DOCTOR** variant is that it shows how little hardware is needed to create ***the illusion of understanding***.

Note : *In 1964-1966,* ***J. Weizenbaum*** *wrote a program for a computer with which one could "converse" in English. Let us give him the floor [2, p.27-33]: "I chose the name "Eliza" for the linguistic analysis program because, like Eliza from the famous "Pygmalion", it could be taught to "speak" better and better. Since one can only talk about "something", i.e. the conversation must have some context, the program was given a two-level organization: the first level includes a linguistic analyzer, the second - a script. The script is a set of rules, approximately the same as those that can be given to an actor to use when improvising on a given topic ... As a first experiment, I gave "Eliza" a script that allows her to play the role (I would even say - parody) of a Rogerian psychotherapist conducting an initial examination of a patient. It is relatively easy to imitate a Rogerian psychotherapist, since his method mainly consists of involving the patient in a conversation by repeating his own statements to him ...*

"The Doctor," as "Eliza," who played the role of a psychiatrist, came to be known, soon became famous at MIT, where the program was born, mainly due to the ease of its demonstration...

The shocks I had to endure during the process of the growth of fame and distribution of the "Doctor" were connected mainly with the following three independent events.

1. *A number of practicing psychiatrists seriously believed that the "Doctor" program could develop into an almost completely automated form of psychiatry...*
2. *I was alarmed to see how quickly the people talking to the "Doctor" came into emotional contact with the computer, how close it was, and how clearly they anthropomorphized it... I had no idea that a very short exposure to a relatively simple machine program could produce such powerful illusions in completely normal people...*
3. *Another unexpected reaction to the Eliza program for me was the spread of the belief that it demonstrated a general solution to the problem of computer understanding of human language..."*

In conclusion, J. Weizenbaum puts forward the following theses: "Firstly, there are fundamental differences between man and machine; secondly, ***there are tasks that should not be entrusted to computers, regardless of whether it is possible to achieve that computers solve them****."*

(1) Winston P. Artificial Intelligence. - M.: Mir, 1980, pp.303-512.
(2) Weizenbaum J. Computer Capabilities and Human Intelligence. From Judgments to Computations. - M.: Radio and Communications, 1982. - 368 p.

From the next step we will start to look at the implementation ***of sorting and searching***.

Step 91.
Sorting. General Provisions

In this step we will provide ***a classification of sorting methods***.

Traditionally, sorting methods are divided into ***internal*** and ***external*** [1, p.10].

Internal methods are those methods that can be applied with acceptable performance only to those data lists that fit entirely in the main (RAM) memory of the processor.

The word ***list*** often refers to a set of records located in main memory.

External methods are those methods that are suitable for data files that are too large to fit in main memory and therefore must be located on external storage devices (tapes, disks, drums) during the sorting process.

In the process of external sorting, a portion of the file is read into the main memory, sorted there, and then written to external devices. This process is repeated several times. Internal methods are used to rearrange the data processed between transfers. So, when we talk about tape sorting, we mean not only the process of reading and writing to these tapes, but also the internal sorting that orders and combines elements from these tapes as they are read.

(1) Lorin G. Sorting and sorting systems. - M.: Nauka, 1983. - 384 p.

In the next step we will look at ***several internal sorting algorithms***.

Step 92.
Sorting items in lists

In this step we will look at ***several internal sorting algorithms***.

1. Let us begin the presentation with a description of a simple ***predicate for checking the ordering of elements in a numeric list***.

Example 1.

```
 (DEFUN TESTP (LAMBDA (LST)
  ; Predicate to test ascending order of ;
  ; elements in a numeric list ;
   (COND ( (NULL LST) T )
         ( (COND ( (EQ (LENGTH LST) 1) T )
                 ( (COND ( (> (CAR LST) (CADR LST))
                                  NIL )
                         (  T  (TESTP (CDR LST)) )) )) )
   )
 ))
```

2. We will begin the description of some internal sorting algorithms with ***internal merge sorting***. The basis of ***the merge process*** is the fundamental idea of ordering data by alternating elements from two or more ordered lists.

Example 2. A function that “merges” the elements ***of two (two-thread merge)*** numeric lists **X** and **Y**, whose elements are sorted in ascending order, into a list “ordered” in ascending order.

```
(DEFUN SORTING1 (LAMBDA (X Y)
   (COND ( (NULL X) Y )
         (   T  (INSERT1 (CAR X) (SORTING1 (CDR X) Y)) )
   )
))
; ------------------------ ;
(DEFUN INSERT1 (LAMBDA (A L)
   (COND ( (NULL L) (LIST A) )
         ( (< A (CAR L)) (CONS A L) )
         (   T  (CONS (CAR L) (INSERT1 A (CDR L))) )
   )
))
```

Test example:

```
$ (SORTING1 '(1 3 5 7) '(4 6 8))
(1 3 4 5 6 7 8)
```

3. ***Exchange sort*** is a general term used to describe a family of memory-saving sorting methods that swap elements of a list if the preceding element ***is ”greater” than*** the preceding one. ***Pairwise exchange***, ***standard exchange***, and ***sifting*** are three simple forms of exchange sort [1, p.24].

Example 3. Sorting the elements of a numeric list in ascending order using the ***standard exchange*** method (also called ***the "bubble method"***).

```
(DEFUN PRIMER (LAMBDA (LST)
   (COND ( (UPORIAD LST) LST )
         (  T  (PRIMER (SORTIROVKA LST)) )
   )
))
; -----------------------------
(DEFUN SORTIROVKA (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (COND ( (EQ (LENGTH LST) 1) LST )
                 ( (COND ( (> (CAR LST) (CADR LST))
                             (CONS (CADR LST)
                                   (SORTIROVKA
                                      (CONS (CAR LST )
                                      (CDDR LST))))
                             )
                         (  T  (CONS (CAR LST)
                                     (SORTIROVKA
                                        (CDR LST))) )) )) )
   )
))
; --------------------------
(DEFUN UPORIAD (LAMBDA (LST)
   (COND ( (NULL LST) T )
         ( (COND ( (EQ (LENGTH LST) 1) T )
                 ( (COND ( (> (CAR LST) (CADR LST)) NIL )
                              (   T  (UPORIAD (CDR LST)) )
                        )
                      )
                )
              )
        )
   ))
```

4. ***Insertion sort*** is the general name for a group of sorting methods based on the sequential insertion of "new" elements into an increasing ordered list. Among them, there are three essentially different methods: ***linear insertion***, ***centered insertion***, and ***binary insertion*** [1, p.33]. These sorting methods differ in the methods for finding a suitable place to insert an element.

The simplest method is ***linear insertion***. As the name suggests, this method treats the existing list as a simple linear list, traversing it element by element from top to bottom until a suitable position for the new element is found.

This method is typically used when a process external to the sort dynamically adds to a list whose elements are all known and which must be kept in an ordered state at all times. The sort is performed each time a new element is received, placing the element in the correct place in the list and facilitating control.

Example 4.

```
   (DEFUN SORTING (LAMBDA (L)
    ; Sort unordered list L ;
   ; using linear insertion method ;
      (COND ( (NULL L) NIL )
            (   T   (INSERT (CAR L) (SORTING (CDR L))) )
      )
   ))
   ; ----------------------- ;
   (DEFUN INSERT (LAMBDA (AL))
   ; The INSERT function adds an element A to an ordered ;
   ; list L so that the order is preserved.
      (COND ( (NULL L) (LIST A) )
            ( (< A (CAR L)) (CONS A L) )
            (   T  (CONS (CAR L) (INSERT A (CDR L))) )
      )
   ))
   ; -------------------------- ;
   (DEFUN RASSTAV (LAMBDA (NEW L)
   ; The RASSTAV function allows elements of the NEW list
   to be inserted into an ordered list L ;
      (COND ( (NULL NEW) L )
            (  T  (INSERT (CAR NEW) (RASSTAV (CDR NEW) L)) )
      )
   ))
```

Test example:

```
   $ (SORTING '(5 4 3 2 1 6))
   (1 2 3 4 5 6)
   $ (RASSTAV '(4 3 1) '(1 2 5 6))
   (1 1 2 3 4 5 6)
```

Example 5.

```
(DEFUN SORTING (LAMBDA (L RANG)
; Sorting of unordered list L. ;
; RANG is a list that specifies the ordering rule ;
; The SORTING, INSERT and LESS-P functions form ;
; a three-level nested recursive structure ;
   (COND ( (NULL L) NIL )
         (   T    (INSERT (CAR L)
                          (SORTING (CDR L) RANG) RANG) )
   )
))
; ---------------------------- ;
(DEFUN INSERT (LAMBDA (AL RANG))
; The INSERT function adds an element A to an ordered ;
; list L so that the ordering is preserved if ;
; the order of any two elements is specified by the ;
; LESS-P predicate. RANG is a list that specifies the ordering rule ;
; The INSERT and LESS-P functions form a two-level nested ;
; iterative structure ;
    (COND ( (NULL L) (LIST A) )
          ( (LESS-P A (CAR L) RANG) (CONS A L) )
          (   T    (CONS (CAR L) (INSERT A (CDR L) RANG)) )
    )
))
; ------------------------------- ;
(DEFUN RASSTAV (LAMBDA (NEW L RANG)
; The RASSTAV function allows elements of the NEW list ;
; to be inserted into an ordered list L. ;
; RANG is a list that specifies the ordering rule ;
   (COND ( (NULL NEW) L )
         (  T  (INSERT (CAR NEW)
                       (RASSTAV (CDR NEW) L RANG) RANG) )
   )
))
; ---------------------------- ;
(DEFUN LESS-P (LAMBDA (A B RANG)
; The LESS-P predicate checks whether element ;
; A occurs before element B according to a certain ;
; order of elements in the RANG list. ;
   (COND ( (NULL RANG) NIL )
         ( (EQ A (CAR RANG)) T )           ; A before ;
         ( (EQ B (CAR RANG)) NIL )         ; B before ;
         (   T    (LESS-P A B (CDR RANG)) )
   )
))
```

Test examples:

```
$ (SORTING '(B A C) '(A B C D E))
(A B C)
$ (INSERT 6 '(5 7 8) '(1 2 3 4 5 6 7 8 9))
(5 6 7 8)
$ (RASSTAV '(B C) '(A B D) '(A B C D E))
(A B B C D)
$ (LESS-P B E '(A B C D E))
T
$ (LESS-P E B '(A B C D E))
NIL
```

Example 6.

```
(DEFUN RANK (LAMBDA (X Y)
; The RANK function orders the list given as ;
; its first argument by rearranging its ;
```

```
   ; elements in the order in which they ;
   ; appear in the list specified as its second ;
   ; argument.
      (COND ( (NULL X) NIL )
            (  T  (CONS (FIRST X Y)
                        (RANK (REMOVEF (FIRST X Y) X) Y)) )
      )
   ))
   ; ---------------------- ;
   (DEFUN FIRST (LAMBDA (X Y)
   ; The FIRST function selects the element of the list specified as the ;
   ; first argument that appears first in the list specified as the ;
   ; second argument. If none of the elements of the first list are ;
   ; contained in the second argument, the first element of the ;
   ; first list is selected. ;
      (COND ( (NULL Y) (CAR X) )
            ( (MEMBER (CAR Y) X) (CAR Y) )
            (  T  (FIRST X (CDR Y)) )
      )
   ))
   ; -------------------------- ;
   (DEFUN REMOVEF (LAMBDA (X LST)
   ; The REMOVEF function removes the first occurrence of element X ;
   ; at the top level from the LST ;
      (COND ( (NULL LST) NIL )
            ( (EQUAL X (CAR LST)) (CDR LST) )
            (  T  (CONS (CAR LST) (REMOVEF X (CDR LST))) )
      )
   ))
```

---

(1) Lopin G. Matching and matching systems. - M.: Hauka, 1983. - 384 s.

In the next step we will look at ***additional sorting functions in* muLISP-85**.

Step 93.
Additional sorting functions

In this step we will introduce ***some functions from* muLISP-85 *that can be used when sorting***.

The muLISP-85 version has two ***modifier functions*** for sorting list elements: **SORT** and **MERGE**.

SORT function syntax:

```
   (SORT LIST TEST)
```

If **TEST** is a permutation of its two arguments, the **SORT** function sorts the elements of **LIST** based on **TEST** and returns the result.

The test **(TEST Object1 Object2)** returns **NIL** if and only if **Object1** is "after" **Object2** or equal to it. In addition, the sorting is correct even if the elements of the **LIST** that **TEST** finds equal will appear in the same order in the result.

The SORT function performs sorting and merging, and the amount of time required is proportional to $\mathbf{log_2\ n}$, where **n** is the length of the **LIST**. For example:

```
   $ (SORT '(5 2 7 3 -4) '<)
   (-4 2 3 5 7)
   $ (SORT '(DOG COW CAT) 'STRING<)
   (CAT COW DOG)
```

The SORT function code looks like this:

```
   (DEGUN SORT (LST TST)
      ( (NULL LST) NIL )
      ( (NULL (CDR LST)) LST )
      ( (SETQ LST (CONS (SORT (SPLIT LST) TST)
                        (SORT LST TST)))
        (MERGE (CDR LST) (CAR LST) TST)
   )
```

Syntax functions **MERGE**:

```
    (MERGE LIST1 LIST2 TEST)
```

If TEST **is** a permutation of the two arguments, the **MERGE** function combines the elements of **LIST1** and **LIST2** based on TEST **and** returns the result.

The (TEST Object1 Object2) function returns **NIL** if and only if **Object1** comes after **Object2** or is equal to it. The union is true if an element of **LIST1** is equal to an element of **LIST2**, then the element of **LIST1** will come before the element of **LIST2** in the result. For example:

```
   $ (MERGE '(2 4 6 8 10) '(3 5 7 11) '<)
   (2 3 4 5 6 7 8 10 11)
```

Here is the function code:

```
   (DEFUN NERGE (LST1 LST2 TST)
      ( (ATON LST1) LST2 )
      ( (ATOM LST2) LST1)
      ( (FUNCALL TST (CAR LST2) (CAR LST1))
          (RPLACD LST2 (MERGE LST1 (CDR LST2) TST)) )
      (RPLACD LST1 (MERGE (CDR LST1) LST2 TST))
   )
```

To estimate the sorting time, you can use the **TIME function in muLISP-83** and higher versions, the syntax of which is

```
    (TIME FL)
```

The TIME function returns the amount of time that has passed since the system clock was set, accurate to hundredths of a second. If the **FL** flag is not **NIL**, the function sets the clock to 0.

The function can be used to determine the most efficient algorithm for solving a particular problem. Resetting the clock to 0 does not affect the operating system's time counter, only the internal **muLISP** time counter. For example:

```
   $ (TIME T) (RECLAIM) (TIME)
   50
```

In the next step we will get acquainted with the organization ***of search in lists***.

Step 94.
Search in lists

In this step we will look at ***the functions that organize the search for elements***.

Here is a small library of LISP functions that allow you to search and replace elements in lists.

Example 1.

```
(DEFUN POSITION (LAMBDA (X LST)
 ; The POSITION function returns the position of atom X in ;
 ; the single-level list LST, counting from the first element in ;
 ; order. ;
 ; If there is no element in the list, then the function ;
 ; returns 0 ;
 ; X is assumed to occur in the LST ;
   (COND ( (NULL LST)       0 )
         ( (EQ X (CAR LST)) 1 )
         (   T  (+ 1 (POSITION X (CDR LST))) )
   )
))
; ---------------------------- ;
(DEFUN FIRST-EQUAL (LAMBDA (X Y)
; The FIRST-EQUAL function returns the first element ;
; in both lists X and Y, otherwise the function ;
; returns NIL ;
   (COND ( (NULL X) NIL )
         ( (NULL Y) NIL )
         ( (MEMBER (CAR X) Y) (CAR X) )
         (   T   (FIRST-EQUAL (CDR X) Y))
   )
))
; ------------------------------- ;
(DEFUN SUBSTRING1 (LAMBDA (LST A B)
; The function replaces all occurrences of atom A in the list LST ;
; with atom B. Note that the replacement is performed only at ;
; the top level of the list LST ;
   (COND ( (NULL LST) NIL )
         ( (EQ (CAR LST) A)
               (CONS B (SUBSTRING1 (CDR LST) A B)) )
         (   T  (CONS (CAR LST)
                      (SUBSTRING1 (CDR LST) A B)) )
   )
))
; ----------------------------------- ;
(DEFUN SUBSTRING2 (LAMBDA (OLD NEW LST)
; The function returns an expression that is the result of ;
; replacing all occurrences of OLD with NEW in LST ;
; (no matter what depth they are at) ;
    (COND ( (EQUAL OLD LST) NEW )
          ( (ATOM LST) LST )
          (    T   (CONS (SUBSTRING2 OLD NEW (CAR LST))
                         (SUBSTRING2 OLD NEW (CDR LST))) )
    )
))
; ------------------------------ ;
(DEFUN SUBSET (LAMBDA (LST1 LST2)
; The function returns T if the list LST1 ;
; is a sublist of the list LST2 ;
   (COND ( (NULL LST1) )
         ( (MEMBER (CAR LST1) LST2)
                   (SUBSET (CDR LST1) LST2) )
         ( T  NIL )
   )
))
```

In the next step, we will provide ***additional functions that can be used when organizing a search***.

Step 95.
Additional search functions

In this step we will look at the **muLISP-85** ***functions that can be used to organize searches***.

Here we will focus on the FIND, FIND-IF, POSITION, and **POSITION-IF** selector functions in the **muLISP-85** version.

Syntax of **FIND** functions:

```
    (FIND OBJECT LIST TEST)
    (FIND-IF TEST LIST)
```

The FIND function performs ***a linear search*** in the **LIST** list for an element for which the check flag with **the OBJECT** object by the **TEST** test is not equal to **NIL** . If the test argument is **NIL** or is not specified, the **FIND** function uses **the EQL** test.

The FIND-IF function examines the **LIST** list to find an element for which the test flag for the **TEST** test is not **NIL**.

For both functions the following is true: if an element that satisfies the test is found, it is returned, otherwise **NIL** is returned.

For example:

```
   $ (FIND 'EAT '(CORN WHEAT OATS RICE) 'FINDSTRING)
   WHEAT
   $ (FIND-IF '(LAMBDA (X) (MINUSP (CDR X))) '((X . 3) (Y .0) (Z . -2/3)))
   (From -0.6666666)
```

Syntax of **POSITION** functions:

```
    (POSITION OBJECT LIST TEST)
    (POSITION-IF TEST LIST)
```

The POSITION function performs ***a linear search*** in the **LIST** list for an element for which the comparison flag with **the OBJECT** object by the **TEST** test is not equal to **NIL**. If the test argument is **NIL** or is not specified, the **POSITION** function uses **the EQL** test.

The POSITION-IF function searches **the LIST** for an element for which the test flag is not **NIL**.

For both functions the following is true: if an element that satisfies the test is found, then the ordinal number of that element, starting from 0, is returned, otherwise **NIL** is returned.

For example:

```
   $ (POSITION '(A B C) '((R S T) (C A B) (A B C)))
   NIL
```

```
    $ (POSITION '(A B C) '((R S T) (C A B) (A B C)) 'EQUAL)
    2
    $ (POSITION-IF 'PLUSP '(-2.5 0  3.7 -5.3))
    2
```

From the next step we will begin to analyze ***the creation of the simplest interpreters***.

Step 96.
The simplest interpreters. Calculating the value of an arithmetic expression

In this step we will consider ***a function that calculates the values of an arithmetic expression***.

The purpose of this and subsequent steps is to show how **LISP** can be used to implement simple interpreters.

Let the list **L** have one of the following structures:

```
    (+ A B)   (- A B)   (* A B)   (/ A B)
```

Here **A** and **B** are either atoms with numerical values or lists of the same form as the list **L**.

A typical list is

```
    (/ (+ A (- B C)) (* (/ D E) F))
```

representing the expression

```
    (A + (B - C)) / ((D / E) * F)
```

Then the function that calculates the value of the arithmetic expression will take the form:

```
    (DEFUN COUNT (LAMBDA (L)
       (COND ( (ATOM L) L)
             ( (EQ (CAR L) +)
                    (+ (COUNT (CADR L))
                          (COUNT (CADDR L))) )
             ( (EQ (CAR L) -)
                 (- (COUNT (CADR L))
                           (COUNT (CADDR L))) )
             ( (EQ (CAR L) *)
                 (* (COUNT (CADR L))
                        (COUNT (CADDR L))) )
             ( (EQ (CAR L) /)
                 (/ (COUNT (CADR L))
                          (COUNT (CADDR L))) )
       )
   ))
```

The result of evaluating the expression

```
    (/ (+ 5 (- 4 2)) (* (/ 6 2) 3))
```

will be as follows:

```
    0.7777777
```

In the next step we will consider ***translating an infix expression into a prefix one***.

Step 97.
The simplest interpreters. Translating an infix expression into a prefix expression

In this step we will introduce ***a function that converts an infix expression to a prefix expression***.

Let's program an INFIX-PREFIX interpreter that converts infix notation of operations (for example, +, -, * and /) of an expression into ***prefix notation*** and returns the value of the expression.

```
(DEFUN INFIX-PREFIX (LAMBDA (L)
 ; Redefinition of arithmetic operation symbols ;
   (PRIN1 "Prefix representation: ")
   (PRINT (PREFIX L))
   (PRIN1 "Calculation result: ")
   (EVAL  (PREFIX L))
))
; -------------------------- ;
(DEFUN PREFIX (LAMBDA (L)
   (ATOM L) L)
   ( (EQ (LENGTH L) 2)         ; Unary operator ;
            (CONS (CAR L) (PREFIX (CADR L))) )
   ( (EQ (LENGTH L) 3)         ; Binary operation ;
            (LIST (CAR L) (PREFIX (CAR L))
                   (PREFIX (CADDR L))) )
   ( PRINT "Error in formula entry!")
))
```

Text example:

```
$ (INFIX-PREFIX '((-2 + 4) * 3))
Prefix representation: (* (+ -2 4) 3)
Calculation result: 6
```

In the next step we will start to look at ***the stages of interpreter design***.

Step 98.
Designing the interpreter. “Spin-up”

In this step we will look at ***general issues of creating interpreters***.

Suppose we want to obtain a translator from a language **L** to a machine **M**, and at the same time write it in the language **L** itself. **L** may be a language specially designed for writing translators, or a high-level language. The desire to write a translator in the same language is quite natural if there is no other suitable language, except, perhaps, autocode. Here we can choose the following line of behavior.

Write a Macroassembler translator for a small subset **L0** of language **L**. This subset should be small enough to be easy to implement and large enough to be used in the next step. The next step is to rewrite the **L0** translator in **L0** itself and debug it. Now we try to **bootstrap *step*** by step to **L**. At each step **i, i=1, 2, ... , n**, the translator for **L[i-1]** is extended to a translator from language **Li** by adding new properties from language **L** . At each step, old parts of the current translator can be rewritten in the new language **Li** to use its new properties. The main reason for extending the **L0 translator to a Ln=L** compiler in several steps instead of one is that at each step we have a more powerful language **Li** for writing the next extension, which greatly simplifies the work [1, pp. 511-512].

"It is sometimes difficult to resist the pressure to include features that "wouldn't be nice to have" as well. There is a real danger that the desire to please everyone will interfere with the goal of getting a coherent design. I have always tried to figure out what the price is for the gain. For example, when considering including a language feature, or adding a special compilation mode for a fairly common construct, one must weigh the benefit against the additional cost of implementing it and simply having it, which makes the system larger. Language designers often fail at this."[2]

Note that some extensions to the **LISP** language have led to the misconception that it is a cumbersome language that takes up megabytes of memory!

Similarly, it is possible to write Interpreters for completely different languages, even for imperative ones such as **Pascal**, although to do this one usually needs ***to rewrite the interpreted language in the form of* S *- expressions*** [1, p.126].

It should also not be forgotten that the **LISP** system is ***interpretive*** rather than compiling: a machine program for calculating a function cannot be compiled outside the process of calculation, before it, because the final form of the function is not known in advance.

---

(1) Gris D. Design of compilers for digital computers. - M.: Mir, 1975. Pp. 367-376, ch.16.
(2) Wirth N. From the development of programming languages to the design of computers // Microprocessor tools and systems, 1989, 4, pp.42-48.
(3) Henderson P. Functional programming: Application and implementation. - M.: Mir, 1983. - 349 p.

From the next step we will begin to consider ***the issues of creating a translator from* LISP *to* muLISPe**.

Step 99.
Designing an Interpreter. LISP in muLISP

In this step we will give ***some considerations for creating an interpreter***.

***Pure* LISP** is a simple subset of **LISP** consisting of:

- basic primitives **CAR, CDR, CONS, EQ** and **ATOM** ;
- control structures using COND, recursion and function superposition;
- list structures containing only atoms and sublists;
- **LAMBDA** primitives for defining new functions.

With such a subset, most of the usual list processing in **LISP** can be done. Pure **LISP** is effectively ***a general-purpose language***.

Pure **LISP** has served as the basis for many theoretical studies in programming. This subset of the **LISP** language has a very simple, regular, recursive structure, which makes it a convenient object for formal analysis [2, pp. 479-480].

The LISP Manual [1] is one of the few programming language manuals that gives a very clear description of the structures that exist at runtime, the structures on which the language implementation is built. It focuses on a complete definition of the interpreter that executes **LISP** programs; this definition takes the form of **LISP** programs for two primitives, **EVAL** and **APPLY**. The structure of the interpreter is so transparent that the entire definition takes up less than two pages.

McCarthy originally defined the interpreter in the meta-notation of the **LISP** ***language*** [3, p.180], where the elegance of the definition of the language is particularly noticeable:

```
evalquote[fn;x] = apply[fn;x;NIL];

apply[fn;x;a] = [ atom[fn] -->
                    [ eq[fn;CAR]  --> caar[x];
                      eq[fn;CDR]  --> cdar[x];
                      eq[fn;CONS] --> cons[car[x];cadr[x]];
                      eq[fn;ATOM] --> atom[car[x]];
                      eq[fn;EQ]   --> eq[car[x];cadr[x]];
                      T           --> apply[eval[fn;a];x;a]
                    ];
                  eq [car[fn];LAMBDA] -->
                     eval [caddr[fn];pairlis[cadr[fn];x;a]];
                  eq [car[fn];LABEL]  -->
                     apply[caddr[fn];x;cons
                             [cons[cadr[fn];caddr[fn]];a]];
                  eq [car[fn];FUNARG] -->
                        apply[cadr[fn];x;caddr[fn]]
                ];

eval [e;a] = [ atom[e]      --> cdr[assoc[e;a]];
               atom[car[e]] -->
                          [ eq[car[e];QUOTE]    --> cadr[e];
                            eq[car[e];COND]     -->
                                            evcon[cdr[e];a];
                            eq[car[e];FUNCTION] -->
                                     list[FUNARG;cadr[e];a];
                            T                   -->
                            apply [car[e];evlis[cdr[e];a];a]
                          ];
               T         --> apply[car[e];evlis[cdr[e];a];a]
             ];

pairlis[x;y;a] = [ null[x] --> a;
                   T       --> cons[cons[car[x];car[y]];
                                   pairlis[cdr[x];cdr[y];a]]
                 ];

assoc[x;a] = [ eq[caar[a];x] --> car[a];
               T             --> assoc[x;cdr[a]]
             ];

evcon[c;a] = [ eval[caar[c];a] --> eval[cadar[c];a];
               T               --> evcon[cdr[c];a]
             ];

evlis[m;a] = [ null[m] --> NIL;
               T       --> cons
                            [eval[car[m];a];evlis[cdr[m];a]]
             ];
```

By comparison, the (revised) Algol 60 report (1963) uses ***1109 syntactic variables*** and the same number of Backus rules, although 73 of them are used only to simplify notation; of the remaining 36 syntactic variables, another half can be eliminated by using the iteration sign. The Pascal report (1971) uses ***197 syntactic variables*** with generalized Backus rules, which use the iteration sign; many of them are again introduced to simplify notation [4, v.2, p.545].

More important in the description of **LISP** is the definition of the structures that exist during the execution of programs. **The A** -list as a means of representing reference environments, the use of property lists to associate function names with their definitions, the descriptions of the representations of atoms, lists, property lists, and numbers during the execution of programs - all this together makes it easy to understand the structure of a program during its execution.

Clarity of definition is important both to the programmer writing **LISP** programs and to the programmer implementing **LISP**. The former needs the language definition to answer subtle questions about its semantics, while the latter needs it as an implementation guide, showing how specific constructs are supposed to work (even though each implementation may differ significantly in detail from the one described in the definition). This **LISP** definition is probably the most widely known definition of a language virtual machine. Unfortunately, few other language definitions have followed this example [2, pp. 479-480].

Writing **LISP** in **LISP** has two important advantages. ***First***, it allows one to define the semantics of the language more strictly, based on a few primitive functions, and ***second***, describing the language in its own language makes it easier to understand and allows one to simultaneously describe both the basic functions of the language and the programming methods it uses. The latter, we hope, will avoid a situation where a novice programmer knows a lot of rules, basic and auxiliary functions and procedures, but is absolutely unable to use his knowledge, since he has no idea yet about the methods of their application.

Defining a language in itself is not such an obvious task. A good explanation of this would be the history of the origin of such an idea [3, p.174-175].

McCarthy relates that while he was thinking about the semantics of the Lisp function EVAL, S. Russell suggested programming it in the same way as other Lisp functions: "... This EVAL was written and published in our paper, when Steve Russell said, "Look, why don't I program this EVAL and you'll get an interpreter," to which I said, "Well, well, you're confusing theory with practice, our EVAL is for reading, not for computing." But he didn't stop there and made an interpreter. So S. Russell translated the EVAL from my paper into 704 machine code, debugged it, and declared the result as a Lisp interpreter, which of course it was..."

Once the idea was put forward, its implementation did not cause any particular difficulties, since in **LISP** data and programs are represented in the same way.

Below we will consider a simple **LISP** interpreter, programmed in itself. It is, of course, different from the interpreters of real systems. Nevertheless, it gives a fairly clear idea of the core of the interpreter and its work, and also shows how easily and elegantly the semantics **of LISP** can be defined and programmed in other languages. In the implementation of the interpreter we will use purely functional programming.

Defining how **LISP** works, using **LISP** as a vehicle, is much like the way dictionaries define words in terms of other, presumably simpler words. Our approach is to reduce **LISP** to a number of basic functions whose definitions are primitives.

Let our interpreter be called **EVAL1** in contrast to the system interpreter **EVAL** that interprets it . Accordingly, we will also name the basic functions that we interpret: **CAR1, CDR1, CONS1, EQUAL1,** and **ATOM1**. In addition, we will use **the LAMBDA1 clause corresponding to the LAMBDA** clause and **the COND1** and **QUOTE1** forms.

T1, NIL1 and integers will be system ***constants.***

During evaluation, it is assumed that the functions called by the user are defined by the **LAMBDA1** clause, which is the **FUN** property of the function name.

Suppose that the variable associations are stored in an association list (A-list) SWQZI.

The generic function EVAL1 can be defined as follows (note that SWQZI is initially an empty list):

```
 (DEFUN EVAL1 (LAMBDA (FORMA SWQZI)
   (COND  ( (ATOM FORMA)    ; FORMA is an atom ;
              (COND  ( (EQ FORMA T1)     T1     )
                     ( (EQ FORMA NIL1)   NIL1   )
                     ( (NUMBERP FORMA)   FORMA )
                     ; Search the SWQZI A-list for the value
```

```
                    of the FORMA atom ;
                    ( (NULL (ASSOC FORMA SWQZI))
                        ; Search in the property list ;
                        ; value of the VALUE property ;
                        ; of the FORMA atom ;
                        (COND ( (NULL (GET FORMA VALUE))
                                   FORMA )
                              (  T  (GET FORMA VALUE) ))
                    )
                    (  T  (CDR (ASSOC FORMA SWQZI)) )
             )
          )
          ; ---------------- ;
          ( (ATOM (CAR FORMA))
            ; The head of the FORMA list is an atom ;
             (COND  ( (EQ (CAR FORMA) QUOTE1)
                          (CADR FORMA) )
                    ; -------------------- ;
                    ( (EQ (CAR FORM) COND1)
                          (EVAL-COND (CDR FORMA) SWQZI) )
                    ; --------------------------------- ;
                    ( (EQ (CAR FORMA) SETQ1)
                    ; Putting the value of the atom ;
                    ; (CAR FORMA) into the property list ;
                         (PUT (CADR FORMA) VALUE
                              (CADDR FORMA)) )
                    ; -------------------- ;
                    ( (EQ (CAR FORMA) GETD1)
                    ; Modeling the GETD function ;
                          (GET (CADR FORMA) FUN)
                    )
                    ; ---------------------------------- ;
                    ; Call basic functions: ;
                    ; CAR1, CDR1, CONS1, ATOM1, EQUAL1, ;
                    ; Call auxiliary functions: ;
                    ; PLUS1,DIFFERENCE1,TIMES1 ;
                    ; GREATERP1,LESSP1 ;
                    ; ---------------------------------- ;
                    ( (MEMBER (CAR FORMA)
                          (QUOTE
                             (CAR1 CDR1 CONS1 ATOM1 EQUAL1
                              PLUS1 DIFFERENCE1 TIMES1
                              GREATERP1 LESSP1)
                          )
                      )
                         (APPLY1 (CAR FORMA)
                                 (EVAL-LIST (CDR FORMA)
                                            SWQZI)
                                 SWQZI)
                    )
                    ; --------------------- ;
                    ( (EQ (CAR FORMA) DEFUN1)
                    ; The LAMBDA definition is placed in the ;
                    ; property list ;
                    ; in the FUN property of the name (CADR FORMA) ;
                         (PUT (CADR FORMA) FUN
                              (CADDR FORMA))
                    )
                    ; ---------------- ;
                    ; Creating a macro ;
                    ( (EQ (CAR FORMA) DEFMACRO1)
                    ; The NLAMBDA definition is placed in the ;
                    ; property list in the NFUN property
                    of the name (CADR FORMA) ;
                          (PUT (CADR FORMA) NFUN
                               (LIST
                                 NLAMBDA1
                                 (CADR (CADDR FORMA))
                                 (CADDR (CADDR FORMA))))
                    )
                    ; -------------------------------- ;
                    ; Calling a user function. ;
                    ; User-called functions ;
                    ; are defined by the LAMBDA1 clause, ;
                    ; which is the FUN property ;
                    ; of the function name ;
                    ; -------------------------------- ;
                    ( (GET (CAR FORMA) FUN)
                      ; The GET function returns the value of
                      the FUN name property (CAR FORMA) ;
                      ; or NIL if there is no such property ;
                           (APPLY1 (GET (CAR FORMA) FUN)
                                   (EVAL-LIST (CDR FORM)
                                                SWQZI)
                                    SWQZI)
                    )
                    ; --------------------------------- ;
                    ; Calling a user function. ;
                    ; User-called functions ;
                    ; are defined by the NLAMBDA1 clause, ;
                    ; which is the NFUN property ;
                    ; of the function name ;
                    ; --------------------------------- ;
                    ( (GET (CAR FORMA) OFFER)
                           (APPLY1
                              (GET (CAR FORMA) OFFER)
                              (NOT-EVAL-LIST (CDR FORMA))
                               SWQZI)
                    )
                    ; ------------------------------ ;
                    (  T  (PRIN1 "Unknown function: ")
                          (PRINT (CAR FORMA)) )
             )
          )
          ; --------------------------------------------- ;
          ; FORMA - a sentence of the form: ;
          ; ((LAMBDA1 Formal_parameters Function_body) ;
          ; Actual_parameters) ;
          ; -------------------------------------------- ;
          ( (EQ (CAAR FORMA) LAMBDA1)
                (APPLY1
                   (CAR FORM)
                   (EVAL-LIST (CDR FORMA) SWQZI) SWQZI)
          )
          ; --------------------------------------------- ;
          ; FORMA - a clause of the form: ;
          ; ((NLAMBDA1 Formal_parameters Function_body) ;
          ; Actual_parameters ) ;
          ; --------------------------------------------- ;
          ( (EQ (CAR FORMA) NLAMBDA1)
                (APPLY1
                   (CAR FORM)
                   (NOT-EVAL-LIST (CDR FORMA)) SWQZI)
          )
          ; -------------------------- ;
          (  T  (PRINT "Syntax error") )
   )
 ))
```

Note that with this definition of the **EVAL-COND** function, the clauses of the **COND1** function are allowed only two elements, i.e. **COND1** allows the following syntax:

```
          (COND1 ( (Condition1) (Preposition1) )
                 ( (Condition2) (Preposition2) )
                      ...           ...
                 ( (ConditionN) (PrepositionN) )
          )
```

Further, if **FORMA** is a list whose first element is a function name, then the expression can be evaluated by getting the **LAMBDA1** clause corresponding to the function name as a property named **FUN** using the **GET** function and applying it using the **APPLY** function.

```
   ( (EQ (CAR FORMA) DEFUN1)
   ; Lambda definition is placed in the property list ;
        (PUT (CADR FORMA) FUN (CADDR FORMA))
   )
   ; ---------------------------------------- ;
   ; Calling a user function. ;
   ; User-called functions are defined
   by the LAMBDA1 clause, which
   is the FUN property of the function name ;
   ; ---------------------------------------- ;
   ( (GET (CAR FORMA) FUN)
          ; The GET function returns the value of
          the FUN name property (CAR FORMA) ;
          ; or NIL if there is no such property ;
          (APPLY1 (GET (CAR FORMA) FUN)
                       (EVAL-LIST (CDR FORM) SWQZI)
                        SWQZI)
   )
```

So far, we have explained rather superficially, almost sloppily, what actually happens when a function is defined and used. We have said that function definitions are somehow remembered, somehow retrieved, and somehow applied to arguments. Now we have refined this description to show how **LISP** works at a deeper level.

We've looked in detail at what **LISP** does when it encounters a user-defined function: the interpreter consults the property list for the lambda definition.

FORMA can be directly **LAMBDA1**-call with actual parameters

```
    ( (LAMBDA1 Formal Parameters Body) Actual Parameters )
```

This situation is handled by the last branch **EVAL1** .

```
   ; ----------------------------------------------- ;
   ; FORMA - a sentence of the form: ;
   ; ( (LAMBDA1 Formal_parameters Function_body) ;
   ; Actual_parameters ) ;
   ; ------------------------------------------------ ;
   ( (EQ (CHANGE SHAPE) LAMBDA1)
         (APPLY1 (FORM CAR) (EVAL-LIST (FORM CDR) SWQZI)
                 SWQZI)
   )
```

The APPLY1 function "applies" functions to argument values. It also defines the basic interpreter functions: **CAR1, CDR1, CONS1, ATOM1, EQUAL1**, and the **LAMBDA1** clause.

```
  (DEFUN APPLY1 (LAMBDA (FUNCT ARGUMENTS SWQZI)
    ; The function returns the result of applying the function ;
    ; FUNCT to the arguments ARGUMENTS ;
      (COND ( (ATOM FUNCT)              ; FUNCT is an atom ;
                 (COND ( (EQ FUNCT CAR1)
                            (COND
                               ( (EQ (CAR ARGUMENTS) NIL1)
                                     NIL1 )
                               (   T   (CAAR ARGUMENTS) ))
                       )
                       ; ------------- ;
                       ( (EQ FUNCT CDR1)
                           (COND ( (EQ (CAR ARGUMENTS) NIL1)
                                      NIL1 )
                                 ( (NULL (CDAR ARGUMENTS))
                                      NIL1 )
                                 ( T  (CDAR ARGUMENTS) ))
                       )
                       ; -------------- ;
                       ( (EQ FUNCT CONS1)
                           (COND ( (EQ (CADR ARGUMENTS) NIL1)
                                     (LIST (CAR ARGUMENTS)) )
                                 (  T  (CONS (CAR ARGUMENTS)
                                        (CADR ARGUMENTS))) )
                       )
                       ; -------------- ;
                       ( (EQ FUNCT ATOM1)
                           (COND ( (ATOM (CAR ARGUMENTS))
                                      T1 )
                                 (  T  NIL1) )
                       )
                       ; --------------- ;
                       ( (EQ FUNCT EQUAL1)
                           (COND ( (EQUAL (CAR ARGUMENTS)
                                          (CADR ARGUMENTS))
                                      T1 )
                                 (  T  NIL1 ))
                       )
                       ; -------------- ;
                       ( (EQ FUNCT PLUS1)
                           (+ (CAR ARGUMENTS)
                                 (CADR ARGUMENTS))
                       )
                       ; -------------------- ;
                       ( (EQ FUNCT DIFFERENCE1)
                           (- (CAR ARGUMENTS)
                                       (CADR ARGUMENTS))
                       )
                       ; --------------- ;
                       ( (EQ FUNCT TIMES1)
                           (* (CAR ARGUMENTS)
                                  (CADR ARGUMENTS))
                       )
                       ; ------------------ ;
                       ( (EQ FUNCT GREATERP1)
                           (COND ( (GREATERP
                                      (CAR  ARGUMENTS)
                                      (CADR ARGUMENTS)) T1 )
                                 (   T   NIL1 ))
                       )
                       ; --------------- ;
                       ( (EQ FUNCT LESSP1)
                           (COND ( (LESSP
                                      (CAR  ARGUMENTS)
                                      (CADR ARGUMENTS)) T1 )
                                 (   T   NIL1 ))
                       )
```

```
                    ; ----------------------------- ;
                    (  T  (APPLY1 (EVAL1 FUNCT SWQZI)
                                  ARGUMENTS SWQZI) )
                )
            )
            ; ---------------------- ;
            ( (EQ (CAR FUNCT) LAMBDA1)
                (EVAL1 (CADDR FUNCT)
                       (SOZDAI-SWQZI (FUNCTION FRAMEWORK)
                                     ARGUMENTS SWQZI))
            )
            ; ----------------------- ;
            ( (EQ (CAR FUNCT) NLAMBDA1)
                (EVAL1 (CADDR FUNCT)
                       (SOZDAI-SWQZI (FUNCTION FRAMEWORK)
                                     ARGUMENTS SWQZI))
            )
            ; -------------------------- ;
            (  T  (PRINT "Syntax error") )
        )
   ))
```

The FUNCT function must be either a basic function known to the interpreter, or a more complex computation defined through a **LAMBDA1** expression.

The SOZDAI-SWQZI function adds connections between formal parameters and actual ones to the OKRUGENIE environment:

```
   (DEFUN SOZDAI-SWQZI (LAMBDA (FORMAL FACTICH OKRUGENIE)
   ; Filling the associative list A of OKRUGENIE connections ;

      (COND
         ( (NULL FORMAL) OKRUGENIE )
         ( (NULL FACTICH) OKRUGENIE )
         (  T  (ACONS (CAR FORMAL) (CAR FACTICH)
                  (SOZDAI-SWQZI
                      (CDR FORMAL) (CDR FACTICH)
                      OKRUGENIE)) )
      )
   ))
```

The second argument of **APPLY-ARGUMENTS** - is a list of values of the arguments of the called function, which is calculated by the **EVAL-LISTOK** function:

```
   (DEFUN EVAL-LIST (LAMBDA (NEWYCHISL SWQZI)
    ; Evaluate the list of argument values ;
    ; of the called function ;
      (COND
         ( (NULL NEWYCHISL) NIL )
         (  T    (CONS (EVAL1 (CAR NEWYCHISL) SWQZI)
                       (EVAL-LIST
                            (CDR NEWYCHISL) SWQZI)) )
      )
   ))
```

Note that the **LISP** language interpreter is formed mainly by two mutually recursive functions **EVAL1** and **APPLY1**, through which the semantics of the language received a formal definition.

Now let's program the dialogue cycle of our LISP interpreter. In the definition below, we will use the symbol **"<--"** as the interpreter prompt. In addition, the symbol **"-->"** will be the sign of the output results.

```
   (DEFUN muLISP (LAMBDA NIL
   ; Dialogue with user '
     (PRINT "Educational muLISP-92 1.01 (06/10/92)")
     (PRINT
     "Copyright (C) McCarthy & Hyvonen & Seppanen & RGPU")
     (TERPRI) (PRIN1 " <-- ")
     ( LOOP
          (SETQ EXPRESSION (READ))
          ( (EQUAL EXPRESSION (LIST SYSTEM1)) 'BYE-BYE )
          (PRIN1 " --> ") (PRINT (EVAL1 EXPRESSION NIL))
          (PRIN1 " <-- ")
     )
  ))
```

---

(1) McCarthy J., Abrahams PW, Edwards J., Hart TP, Levin MI LISP 1.5 Programmer's Manual. - MIT Press, Cambridge, Massachusetts, 1965.
(2) Pratt T. Programming Languages. Development and Implementation. - Moscow: Mir, 1979. - 574 p.
(3) Hyvönen E., Seppänen J. The World of Lisp. In 2 volumes. Vol. 2: Programming Methods and Systems. - Moscow: Mir, 1990. - 319 p.
(4) Bauer F.L., Goos G. Computer Science. Introductory Course: In 2 Parts. Part 1. translated from German. - Moscow: Mir, 1990. - 336 p.; Part 2. translated from German. - M.: Mir, 1990. - 423 p.

In the next step we will provide the text of the educational **LISP** ***interpreter in*** **muLISPe**.

Step 100.
Designing the interpreter. Text of the training interpreter

In this step we will provide ***the text of the training interpreter***.

Program. Educational **LISP** interpreter in the **muLISP** dialect.

```
  (PROG1 "" (PUTD DEFUN (QUOTE (NLAMBDA (FUNC DEF)
                                (PUTD FUNC DEF) FUNC ))))
  ; ------------------------------------- ;
  ; LISP interpreter in the muLISP dialect ;
  ; ------------------------------------- ;
  (DEFUN muLISP (LAMBDA NIL
  ; Dialogue with the user ;
     (PRINT "Educational muLISP-92 1.01 (06/10/92)")
     (PRINT
     "Copyright (C) McCarthy & Hyvonen & Seppanen & RGPU")
     (TERPRI) (PRIN1 " <-- ")
     ( LOOP
          (SETQ EXPRESSION (READ))
          ( (EQUAL EXPRESSION (LIST SYSTEM1)) 'BYE-BYE )
          (PRIN1 " --> ") (PRINT (EVAL1 EXPRESSION NIL))
          (PRIN1 " <-- ")
     )
  ))
  ; ------------------------------ ;
  (DEFUN EVAL1 (LAMBDA (FORM SWQZI)
    (COND ( (ATOM FORMA)      ; FORMA is an atom ;
             (COND  ( (EQ FORMA T1)    T1    )
                    ( (EQ FORMA NIL1)  NIL1  )
                    ( (NUMBERP FORM) FORM )
                    ; Search the SWQZI A-list for the value
                    of the FORMA atom ;
```

```
                ( (NULL (ASSOC FORM SWQZI))
                    ; Search in the property list ;
                    ; value of the VALUE property ;
                    ; of the FORMA atom ;
                    (COND ( (NULL (GET FORMA VALUE))
                                FORM )
                          (  T  (GET FORMA VALUE) ))
                )
                ( T (CDR (ASSOC FORMA SWQZI)) )
        )
      )
      ; ---------------- ;
      ( (ATOM (CAR SHAPE))
        ; The head of the FORMA list is an atom ;
         (COND  ( (EQ (CAR FORMA) QUOTE1)
                      (CADR FORM) )
                ; -------------------- ;
                ( (EQ (CAR FORM) COND1)
                      (EVAL-COND (CDR FORMA) SWQZI) )
                ; --------------------------------- ;
                ( (EQ (CAR SHAPE) SETQ1)
                ; Putting the value of the atom ;
                ; (CAR FORMA) into the property list ;
                     (PUT (CADR FORMA) VALUE
                          (CADDR FORMA)) )
                ; -------------------- ;
                ( (EQ (CAR FORMA) GETD1)
                ; Modeling the GETD function ;
                      (GET (CADR FORMA) FUN)
                )
                ; ---------------------------------- ;
                ; Call basic functions: ;
                ; CAR1, CDR1, CONS1, ATOM1, EQUAL1, ;
                ; Call auxiliary functions: ;
                ; PLUS1,DIFFERENCE1,TIMES1 ;
                ; GREATERP1,LESSP1 ;
                ; ---------------------------------- ;
                ( (MEMBER (CAR FORMA)
                      (QUOTE
                         (CAR1 CDR1 CONS1 ATOM1 EQUAL1
                          PLUS1 DIFFERENCE1 TIMES1
                          GREATERP1 LESSP1)
                      )
                  )
                     (APPLY1 (CAR FORMA)
                             (EVAL-LIST (CDR FORM)
                                        SWQZI)
                              SWQZI)
                )
                ; --------------------- ;
                ( (EQ (CAR SHAPE) DEFUN1)
                ; The LAMBDA definition is placed in the ;
                ; property list ;
                ; in the FUN property of the name (CADR FORMA) ;
                     (PUT (FRAME FORM) FUN
                          (CADDR FORMA))
                )
                ; ---------------- ;
                ; Creating a macro ;
                ( (EQ (CAR FORMA) DEFMACRO1)
                ; The NLAMBDA definition is placed in the ;
                ; property list in the NFUN property
                of the name (CADR FORMA) ;
                      (PUT (CADR FORMA) GIVEN
                           (LIST
                              NLAMBDA1
                              (CADR (CADDR FORMA))
                              (CADDR (CADDR FORMA))))
                 )
                 ; -------------------------------- ;
                 ; Calling a user function. ;
                 ; User-called functions ;
                 ; are defined by the LAMBDA1 clause, ;
                 ; which is the FUN property ;
                 ; of the function name ;
                 ; -------------------------------- ;
                 ( (GET (CAR FORMA) FUN)
                   ; The GET function returns the value of
                   the FUN name property (CAR FORMA) ;
                   ; or NIL if there is no such property ;
                        (APPLY1 (GET (CAR FORMA) FUN)
                                (EVAL-LIST (CDR FORM)
                                             SWQZI)
                                 SWQZI)
                 )
                 ; --------------------------------- ;
                 ; Calling a user function. ;
                 ; User-called functions ;
                 ; are defined by the NLAMBDA1 clause, ;
                 ; which is the NFUN property ;
                 ; of the function name ;
                 ; --------------------------------- ;
                 ( (GET (CAR FORMA) OFFER)
                        (APPLY1
                           (GET (CAR FORMA) OFFER)
                           (NOT-EVAL-LIST (CDR FORMA))
                            SWQZI)
                 )
                 ; ------------------------------ ;
                 (  T  (PRIN1 "Unknown function: ")
                       (PRINT (FORM CARD)) )
            )
         )
         ; -------------------------------------------- ;
         ; FORMA - a sentence of the form: ;
         ; ((LAMBDA1 Formal_parameters Function_body) ;
         ; Actual_parameters) ;
         ( (EQ (CHANGE SHAPE) LAMBDA1)
              (APPLY1
                 (CAR FORM)
                 (EVAL-LIST (CDR FORMA) SWQZI) SWQZI)
         )
         ; --------------------------------------------- ;
         ; FORMA - a clause of the form: ;
         ; ((NLAMBDA1 Formal_parameters Function_body) ;
         ; Actual_parameters ) ;
         ; --------------------------------------------- ;
         ( (EQ (CAR FORMA) NLAMBDA1)
              (APPLY1
                 (CAR FORM)
                 (NOT-EVAL-LIST (CDR FORM)) SWQZI)
         )
         ; -------------------------- ;
         (  T  (PRINT "Syntax error") )
  )
))
; ----------------------------------------- ;
(DEFUN APPLY1 (LAMBDA (FUNCT ARGUMENTS SWQZI)
; The function returns the result of applying the function ;
; FUNCT to the arguments ARGUMENTS ;
```

```
      (COND ( (ATOM FUNCT)             ; FUNCT - atom ;
                 (COND ( (EQ FUNCT CAR1)
                            (COND
                               ( (EQ (CAR ARGUMENTS) NIL1)
                                     NIL1 )
                               ( T (SEVEN ARGUMENTS) ))
                       )
                       ; ------------- ;
                       ( (EQ FUNCT CDR1)
                           (COND ( (EQ (CAR ARGUMENTS) NIL1)
                                      NIL1 )
                                 ( (NULL (CDAR ARGUMENTS))
                                      NIL1 )
                                 ( T  (CDAR ARGUMENTS) ))
                       )
                       ; -------------- ;
                       ( (EQ FUNCT CONS1)
                           (COND ( (EQ (CADR ARGUMENTS) NIL1)
                                     (LIST (CAR ARGUMENTS)) )
                                 (  T  (CONS (CAR ARGUMENTS)
                                        (CADR ARGUMENTS))) )
                       )
                       ; -------------- ;
                       ( (EQ FUNCT ATOM1)
                           (COND ( (ATOM (CAR ARGUMENTS))
                                      T1 )
                                 (  T  NIL1) )
                       )
                       ; --------------- ;
                       ( (EQ FUNCT EQUAL1)
                           (COND ( (EQUAL (CAR ARGUMENTS)
                                          (CADR ARGUMENTS))
                                      T1 )
                                 (  T  NIL1 ))
                       )
                       ; -------------- ;
                       ( (EQ FUNCT PLUS1)
                           (PLUS (CAR ARGUMENTS)
                                 (CADR ARGUMENTS))
                       )
                       ; -------------------- ;
                       ( (EQ FUNCT DIFFERENCE1)
                           (DIFFERENCE (CAR ARGUMENTS)
                                       (CADR ARGUMENTS))
                       )
                       ; --------------- ;
                       ( (EQ FUNCT TIMES1)
                           (TIMES (CAR ARGUMENTS)
                                  (CADR ARGUMENTS))
                       )
                       ; ------------------ ;
                       ( (EQ FUNCT GREATERP1)
                           (COND ( (GREATERP
                                      (CAR  ARGUMENTS)
                                      (CADR ARGUMENTS)) T1 )
                                 (   T   NIL1 ))
                       )
                       ; --------------- ;
                       ( (EQ FUNCT LESSP1)
                           (COND ( (LESSP
                                      (CAR  ARGUMENTS)
                                      (CADR ARGUMENTS)) T1 )
                                 (   T   NIL1 ))
                       )
                       ; ----------------------------- ;
```

```
                     (  T  (APPLY1 (EVAL1 FUNCT SWQZI)
                                   ARGUMENTS SWQZI) )
                )
           )
           ; ---------------------- ;
           ( (EQ (CAR FUNCT) LAMBDA1)
                (EVAL1 (CADDR FUNCT)
                       (SOZDAI-SWQZI (FUNCTION FRAMEWORK)
                                     ARGUMENTS SWQZI))
           )
           ; ----------------------- ;
           ( (EQ (CAR FUNCT) NLAMBDA1)
                (EVAL1 (CADDR FUNCT)
                       (SOZDAI-SWQZI (FUNCTION FRAMEWORK)
                                     ARGUMENTS SWQZI))
           )
           ; --------------------------- ;
           (   T  (PRINT "Syntax error") )
     )
  ))
  ; ------------------------------------- ;
  (DEFUN EVAL-COND (LAMBDA (VETVI CONTEKST)
  ; Definition of semantics of conditional operator COND1 ;
      (COND ( (NULL VETVI) NIL1 )
            ( (NOT (EQ (EVAL1 (CAAR VETVI) CONTEKST) NIL1))
                       (EVAL1 (VETCHILMET) CONTEXT) )
            ( T (EVAL-COND (CDR VETVI) CONTEKST) )
      )
  ))
  ; ---------------------------------------- ;
  (DEFUN EVAL-LIST (LAMBDA (NEWYCHISL SWQZI)
  ; Calculate the list of argument values ;
  ; of the called function ;
     (COND ( (NULL NEWYCHISL) NIL )
           (  T   (CONS (EVAL1 (CAR NEWYCHISL) SWQZI)
                        (EVAL-LIST
                           (CDR NEWYCHISL) SWQZI)) )
     )
  ))
  ; -------------------------------------- ;
  (DEFUN NOT-EVAL-LIST (LAMBDA (NEWYCHISL)
  ; Prevent evaluation of the list of argument values of the called function ;
     (COND ( (NULL NEWYCHISL) NIL )
           (  T   (CONS (CAR NEWYCHISL)
                        (NOT-EVAL-LIST
                            (CDR NEWYCHISL))) )
     )
  ))
  ; -------------------------------------------------- ;
  (DEFUN SOZDAI-SWQZI (LAMBDA (FORMAL FACTICH OKRUGENIE)
  ; Filling the associative list A of OKRUGENIE connections ;
     (COND ( (NULL FORMAL) OKRUGENIE )
           ( (NULL FACT) OKRUGENIE )
           (  T  (ACONS (CAR FORMAL) (CAR FACTICH)
                        (SOZDAI-SWQZI (CDR FORMAL)
                                      (CDR FACTICH)
                                      OKRUGENIE)) )
     )
  ))
  ; ---------------------------- ;
  (DEFUN ACONS (LAMBDA (X Y ALIST)
  ; Adding a dotted pair (X . Y) to the beginning of the association list ALIST ;
       (CONS (CONS X Y) ALIST)
  ))
```

Note that embedding the “interpretation layer” is the first step toward creating original control structures. The interpreter, located between the standard **LISP** procedures and the user program, allows the programmer to perform complex "surgical" operations if there is no desire to adapt to the existing version of the interpreter, which is usually hidden in the implementation at the assembly language level.

The programmer can invent new control structures because the material from which such structures are built is readily available to him. This is the way in which many high-level languages have been implemented. Usually, however, this comes at a high price. An additional level of interpretation generally results in ***a significant reduction in the speed of the program***.

Note, however, that interpreters are important for three very important reasons.

First, they can provide a convenient interactive environment. Many new users find that the interactive environment is more convenient than the compiler.

Second, language interpreters provide excellent interactive debugging capabilities. Even veteran programmers resort to the help of a language interpreter when debugging difficult programs because it allows dynamic setting of variables and conditions.

Third, most database query languages operate in interpreted mode.

In the next step we will give some general guidelines for debugging programs.

Step 101.
General information on debugging programs

In this step we will provide ***some general information about debugging programs***.

Here are some test examples for our interpreter: unfortunately (!), they did not allow us to find errors:

```
1. <-- ((NLAMBDA1 (X) (CDR1 X)) (1 (TIMES1 1 2) 4))
   --> ((TIMES1 1 2) 4)
   <-- ((NLAMBDA1 (X) (CAR1 X)) ((TIMES1 1 2) (PLUS1 1 4)))
   --> (TIMES 1 2)
2. <-- (DEFUN1 SUMMA (LAMBDA1 (LST) (COND1 ((EQUAL1 LST
   NIL1) 0) (T1 (PLUS1 (CAR1 LST) (SUMMA (CDR1 LST))))))
   --> (LAMBDA1 (LST) (COND1 ((EQUAL1 LST NIL1) 0) (T1
   (PLUS1 (CAR1 LST) (SUMMA (CDR1 LST))))))
   <-- (SUMMA (QUOTE1 (1 2 3)))
   --> 6
3. <-- (DEFUN1 FACT (LAMBDA1 (X) (COND1 ((EQUAL1 X 1) 1)
   (T1 (TIMES1 X (FACT (DIFFERENCE1 X 1)))))))
   --> (LAMBDA1 (X) (COND1 ((EQUAL1 X 1) 1) (T1 (TIMES1
   X (FACT (DIFFERENCE1 X 1))))))
   <-- (FACT 5)
   --> 120
4. <-- (DEFUN1 TEST (LAMBDA1 (X Y) (CONS1 X Y)))
   --> (LAMBDA1 (X Y) (CONS1 X Y))
   <-- (TEST A B)
   --> (A . B)
5. <-- (DEFUN1 COPY (LAMBDA1 (E) (COND1 ((ATOM1 E) E) (T1
   (CONS1 (COPY (CAR1 E)) (COPY (CDR1 E)))))))
   --> (LAMBDA1 (E) (COND1 ((ATOM1 E) E) (T (CONS1 (COPY
   (CAR1 E)) (COPY (CDR1 E))))))
   <-- (COPY (QUOTE1 (A B C)))
   --> (A B C)
6. <-- (DEFMACRO1 AAA (NLAMBDA1 (G) (CAR1 G)))
   --> (NLAMBDA1 (G) (CAR1 G))
```

```
     <-- (AAA ((TIMES1 2 3) 4 5))
     --> (TIMES1 2 3)
```

Further, you can further expand the capabilities of the constructed interpreter, using your own mechanism for defining functions.

For example, let's define the **NULL1** function through the system primitives:

```
  <-- (DEFUN1 NULL1 (LAMBDA1 (L) (EQUAL1 L NIL1)))
  --> (LAMBDA1 (L) (EQUAL1 L NIL1))
```

and let's use it right away:

```
  7. <-- (DEFUN1 MEMBER1 (LAMBDA1 (AL LST) (COND1 ((NULL1
     LST) NIL1) ((EQUAL1 AL (CAR1 LST)) LST) (T1 (MEMBER1
     AL (CDR1 LST))))))
     -->  (LAMBDA1 (AL LST) (COND1 ((NULL1 LST) NIL1)
     ((EQUAL1 AL (CAR1 LST)) LST) (T1 (MEMBER1 AL (CDR1
     LST)))))
     <-- (MEMBER1 (QUOTE1 B) (QUOTE1 (A B C)))
     --> (BC)
  8. <-- (DEFUN1 DEL (LAMBDA1 (A L) (COND1 ((NULL1 L) NIL1)
     ((EQUAL1 (CAR1 L) A) (CDR1 L)) (T1 (CONS1 (CAR1 L)
     (DEL A (CDR1 L)))))))
     --> (LAMBDA1 (A L) (COND1 ((NULL1 L) NIL1) ((EQUAL1
     (CAR1 L) A) (CDR1 L)) (T1 (CONS1 (CAR1 L) (DEL A (CDR1
     L)))))
     <-- (DEL 5 (QUOTE1 (3 4 5)))
     --> (3 4)
  9. <-- (DEFUN1 EVERY (LAMBDA1 (L) (COND1 ((NULL1 L) NIL1)
     ((NULL1 (CDR1 L)) L) (T1 (CONS1 (CAR1 L) (EVERY (CDR1
     (CDR1 L))))))))
     --> (LAMBDA1 (L) (COND1 ((NULL1 L) NIL1) ((NULL1 (CDR1
     L)) L) (T1 (CONS1 (CAR1 L) (EVERY (CDR1 (CDR1 L)))))))
     <-- (EVERY (QUOTE1 (A P R O L D)))
     (A R L)
 10. <-- (DEFUN1 APPEND1 (LAMBDA1 (L1 L2) (COND1 ((NULL1 L1)
     L2) ( T1 (CONS1 (CAR1 L1) (APPEND1 (CDR1 L1) L2)))))))
     --> (LAMBDA1 (L1 L2) (COND1 ((NULL1 L1) L2)  (T1 (CONS1
     (CAR1 L1) (APPEND1 (CDR1 L1) L2))))))
     <-- (APPEND1 (QUOTE1 (A P R)) (QUOTE1 (O L D)))
     (A P R O L D)
```

After checking, you may be interested in three questions:

- what is ***a "test case"*** ;
- why the tests ***"unfortunately"*** did not allow finding errors in the program;
- why these tests were chosen and not some others and based on what considerations they should be chosen.

The answer to the third question is: when choosing these tests, we were not guided by any theoretical principles, these were the first functions that came to mind. In this regard, you are given the task: ***to identify the shortcomings of the proposed test system and improve it***.

We will answer the first two questions below (see more detailed discussion in [1]).

We will say that "***there is an error in a program*** if its execution does not meet the user's expectations" [2, 3].

Let us recall that when solving problems using a computer, debugging ***programs*** is usually understood as one of the stages of the solution, during which the computer is used to detect and correct ***errors*** in the program;

during debugging, the programmer wants to achieve a certain degree of confidence that his program is appropriate for its purpose and does not do what it is not intended for.

The components of the debugging process can be schematically represented in the "Backus-Naur form":

```
                                  ¦ Process ¦ ¦ Process ¦
  [Debugging process] ::=         ¦ search  ¦ ¦ fix     ¦
                                  ¦ errors  ¦ ¦ errors  ¦

       ¦ Process ¦ ¦ Process ¦ ¦ Process ¦
       ¦ search ¦ ::= ¦ testing ¦ ¦ localization ¦
       ¦ errors ¦ ¦ programs ¦ ¦ errors ¦
```

A novice programmer, as a rule, overestimates his capabilities and, when developing a program, assumes that his program will be error-free. And when speaking about a program he has just compiled, he is ready to assure that it is 99% correct, and all that remains is to run it once (!) on a computer with some (!) initial data for greater certainty. Naturally, every incorrect result, every error found amazes him and is considered, of course, the last. As a result of this approach, obtaining reliable results from a computer for a compiled program is postponed for a long and indefinite period.

It turns out that it is practically impossible for a sufficiently complex program to quickly find and eliminate all the errors in it. The difficulties of programming and debugging are emphasized by the following popular aphorism among programmers: " ***Every*** program has at least one error." Therefore, it can be said that the presence of errors in a newly developed program is a completely normal and natural phenomenon. And a completely abnormal, out-of-the-ordinary fact is the absence of errors in a program that has not yet been thoroughly tested and debugged (of course, we are talking about fairly complex programs here!).

Therefore, it is reasonable to prepare for the detection of errors at the debugging stage already during the development of a program at the algorithmization and programming stages and take preventive measures to prevent them.

For more information on debugging programs, see the Debugging section.

From the next step we will begin to consider the relationship of the **LISP** ***programming language with other languages*** .

Step 102.
LISP and TURBO Pascal. Binary trees with labeled leaves

In this step we will look at ***creating binary trees with labeled leaves in*** **TURBO Pascal** language.

An abstract data type in **Pascal** that is of considerable theoretical and practical interest is the binary list, which is recursively defined as follows:

```
   a binary list is either an atomic binary list (symbol),
                    or an ordered pair of binary lists.
```

A binary list is graphically represented as a binary tree with labeled leaves:

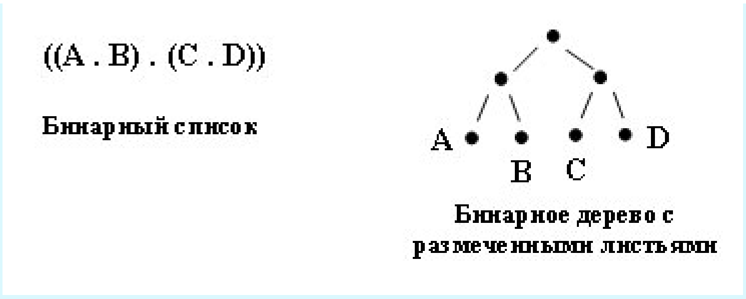


Fig. 1. Graphical representation

In this regard, everywhere below, instead of the term " ***binary list*** ", we will use the term " ***binary tree with marked leaves*** ".

Let us list the main operations on the binary tree data type with marked leaves [2]:

- joining two trees into one (designated **CONS**);
- left subtree selection (denoted by **CAR**);
- selection of the right subtree (denoted as **CDR**).

The CAR and **CDR** operations are partial inversions of the **CONS** operation;

- the predicate **ATOM**, which allows us to distinguish between atomic and "real" trees;
- predicate **EQ**, which allows us to compare the "leaves" of a tree.

Note that labeled binary trees with the join operation form ***a groupoid*** (groupoid of labeled trees). ***A groupoid*** is an algebra with one arbitrary operation.

Moreover, it turns out that binary trees with labeled leaves are the simplest data type of the **LISP** language, namely: **S** -expressions. Recall that ***the set of* S*-expressions*** is defined recursively as the minimal set such that the atoms are **S** -expressions, and if **S1** and **S2** are **S** -expressions, then the pair **(S1, S2)** is also **an S-** expression.

Thus, it is shown that binary marked trees are typical objects of the **LISP** programming language, namely **S** -expressions. Let us proceed to the definition of binary trees with marked leaves in the **Pascal** language.

To do this, we use the following type definition:

```
      type  TipElement = Char;
            Lisp       = ^LispEl;
            LispEl     = Record
                           Case Tag: (Single,Pair) of
                                      { А т о м  }
                              Single: (Leaf : TipElement);
                                      { Dotted pair }
                              Pair   : (Left : Lisp;
                                       Right: Lisp)
                           End;
```

To work with **S** -expressions, we define the following five basic functions: **CAR, CDR, ATOM, EQUAL** and **CONS** [1, pp. 204-209]:

- ***constructor function***

```
   FUNCTION CONS (X,Y: Lisp): Lisp;
    { Construct an S-expression from given S-expressions X and Y }
      var   p: Lisp;
    BEGIN
      New (p); p^.Tag := Pair; p^.Left := X; p^.Right := Y;
      CONS := p
   END;
```

- ***selector functions***

```
FUNCTION   CAR (X: Lisp): Lisp;
 { Extract first component of S-expression X }
BEGIN
   CAR := X^.Left
END ;
{ ---------------------------- }
FUNCTION   CDR (X: Lisp): Lisp;
 { Extract the second component of the S-expression X }
BEGIN
   CDR := X^.Right
END;
```

- ***discriminator function***

```
FUNCTION   ATOM (X: Lisp): Boolean;
 { Test argument type X }
BEGIN
   ATOM := (X^.Tag = Single)
END;
```

- ***function for checking the identity of atomic S-expressions***

```
FUNCTION   EQ (X,Y: Lisp): Boolean;
 { Test for equality of two atomic S-expressions X and Y }
BEGIN
   EQ := (X^.Leaf = Y^.Leaf)
END;
```

Notes .

1. *Using basis functions, it is easy to construct a function for checking the identity* ***of any S-expressions****:*

```
FUNCTION  EQUAL (X,Y: Lisp): Boolean;
{ Checking for the equality of S-expressions X and Y }
BEGIN
   If  (ATOM (X)) OR (ATOM (Y))
      then  If  (ATOM (X)) AND (ATOM (Y))
                then  EQUAL := EQ (X,Y)
                else  EQUAL := FALSE
      else  If  EQUAL (CAR (X),CAR (Y))
                then  EQUAL := EQUAL (CDR (X),CDR (Y))
                else  EQUAL := FALSE
END;
```

2. *The caution required when using the* ***CAR*** *and* ***CDR*** *functions and the union of types in general is shown in the example of the* ***FIRSTATOM*** *function: [1, p.205]*

```
FUNCTION   FIRSTATOM (X: Lisp): TipElement;
 { Definition of the first atom of S-expression X }
BEGIN
   If   ATOM (X)
       then   FIRSTATOM := X^.Leaf
       else   FIRSTATOM := FIRSTATOM (CAR (X))
 END ;
```

3. *A binary tree with labeled leaves can be constructed operationally, i.e. described in its final form using some formula. The records of such composite objects are called* ***terms****.*

For example, a term for a binary tree with labeled leaves composed of objects ***A, B, C, D***

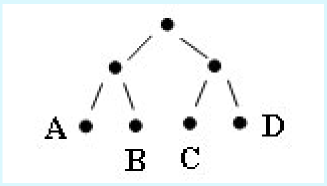


Fig.2. Binary tree with labeled leaves

has the form:

```
     CONS (CONS (MAKEATOM('A'),MAKEATOM('B')),
                CONS (MAKEATOM('C'),MAKEATOM('D')))
```

Example: Representation of LISP dotted pairs as binary trees with labeled leaves.

```
   PROGRAM   LISPAS;      {$A-}
  { Representation of LISP dot pairs as binary }
  { trees with labeled leaves }
      type   TipElement = Char;
             Lisp       = ^LispEl;
             LispEl     = Record
                            Case Tag: (Single,Pair) of
                                       { А т о м  }
                               Single: (Leaf : TipElement);
                                       { Dotted pair }
                               Pair  : (Left : Lisp;
                                        Right: Lisp)
                            End;
             Pod = String [23];
     { ---------------------------------------------------- }
      var    Root1 : Lisp;    { Pointer to dotted pair }
            Root2 : Lisp;    { Pointer to dotted pair }
            Strk : Stroka; { String - dotted pair }
            Result: Stroka; { Intermediate form of dotted pair }
            i : Integer; { Auxiliary variable }
  { ------------------------------------------------------- }
   PROCEDURE   Enter ( var T: Lisp);
    { Construction of binary tree, }
   { corresponding to S-expression of LISP language }
      var    X: Char;
    BEGIN
      Write (' '); X := Strk[i]; i := i + 1;
     { Put the dotted pair element X into the binary tree }
      If    X='#'
          then    begin
                    New (T); T^.Tag := Pair;
                   Enter (T^.Left); Enter (T^.Right)
                 end
         else  begin
                   New (T); T^.Tag := Single;
                   T^.Leaf := X
                 end
   END ;
   { ------------------------------------------------- }
   PROCEDURE   PrintTree (W: Lisp; l: Integer);
   { Output the binary tree corresponding to }
  { dotted pair W }
     var   i: Integer;
   BEGIN
     If   W^.Tag <> Single
         then   begin
                  PrintTree (W^.Right,l+1);
                  For i:=1 to l do  Write ('   ');
                  Writeln('#');
                  PrintTree (W^.Left,l+1)
                end
        else  begin
                  For i:=1 to l do  Write ('   ');
                  Writeln (W^.Leaf)
                end
  END ;
  { ------------------------------------ }
  PROCEDURE   Convert (Strk: Stroka;
                             var Result: Stroka);
      var   k : Integer;    { Parameter Piece }
          Ch: Char;       { Generalized symbol }
  BEGIN
     Result := '';
     For k:=1 to Length (Strk)  do
        begin
           Ch := Strk[k];
           If  Ch='('
              then  Result := Result + '#'
              else  If  (Ch<>')') AND (Ch<>'.')
                                 AND (Ch<>' ')
                     then  Result := Result + Ch
        end
  END ;
  { -------------------------------- }
  FUNCTION   ATOM (X: Lisp): Boolean;
   { Argument type check }
  BEGIN
     ATOM := (X^.Tag = Single)
  END ;
  { ---------------------------- }
  FUNCTION   CAR (X: Lisp): Lisp;
   { Extract the first component of an S-expression }
  BEGIN
     CAR := X^.Left
  END ;
  { ---------------------------- }
  FUNCTION   CDR (X: Lisp): Lisp;
  { Extract the second component of the S-expression }
  BEGIN
     CDR := X^.Right
  END ;
  { -------------------------------- }
  FUNCTION   EQ (X,Y: Lisp): Boolean;
   { Test equality of two atomic S-expressions }
  BEGIN
     EQ := (X^.Leaf = Y^.Leaf)
  END ;
  { ------------------------------ }
  FUNCTION CONS (X,Y: Lisp): Lisp;
   { Constructing an S-expression from given S-expressions X and Y }
     var   p: Lisp;
   BEGIN
     New (p); p^.Tag := Pair; p^.Left := X; p^.Right := Y;
     CONS := p
  END;
```

```
{ ----------------------------------- }
 FUNCTION  EQUAL (X,Y: Lisp): Boolean;
 { Checking for the equality of S-expressions X and Y }
 BEGIN
    If  (ATOM (X)) OR (ATOM (Y))
       then  If  (ATOM (X)) AND (ATOM (Y))
                then  EQUAL := EQ (X,Y)
                else  EQUAL := FALSE
       else  If  EQUAL (CAR (X),CAR (Y))
                then  EQUAL := EQUAL (CDR (X),CDR (Y))
                else  EQUAL := FALSE
 END;
{ --------------------------------------- }
 FUNCTION  MAKEATOM (C: TipElement): Lisp;
    var  h: Lisp;
 BEGIN
    New (h); h^.Tag := Single; h^.Leaf := C; MAKEATOM := h
 END;
{ ---------------------------------- }
 FUNCTION  VAL (A: Lisp): TipElement;
 BEGIN
    VAL := A^.Leaf
 END ;
 { ---------------------------------------- }
 FUNCTION   FIRSTATOM (X: Lisp): TipElement;
  { Definition of the first atom of the S-expression }
 BEGIN
    If   ATOM (X)
        then   FIRSTATOM := X^.Leaf
        else   FIRSTATOM := FIRSTATOM (CAR (X))
  END ;
 { ------------------------------------------ }
 BEGIN
    Writeln
    ('Let's construct binary trees with labeled leaves');
    Writeln('by given terms...');
    PrintTree
       (CONS (CONS (MAKEATOM('A'),MAKEATOM('B')),
              CONS (MAKEATOM('C'),MAKEATOM('D'))),0);
    Writeln;
    Writeln(' ------------------------------------ ');
    PrintTree (CONS (MAKEATOM('A'),
                       CONS (MAKEATOM ('B'),
                             CONS (MAKEATOM('C'),
                                   MAKEATOM('D')))),0);
    Writeln(' ------------------------------------ ');
    Writeln
    ('Let's construct a binary tree with labeled leaves');
    Writeln('given a Lisp dotted pair...');
    Writeln('Enter first dot pair...');
    i := 1; ReadLn (Strk); Convert (Strk,Result);
    Strk := Result; Root1 := Nil; Enter (Root1);
    Writeln; PrintTree (Root1,0); Writeln;
    Writeln(' ------------------------------------- ');
    Writeln('Enter second dot pair...');
    i := 1; ReadLn (Strk); Convert (Strk,Result);
    Strk := Result; Root2 := Nil; Enter (Root2);
    Writeln; PrintTree (Root2,0); Writeln;
    Writeln('----------------------------------------');
    Writeln ('Demonstration actions functions CONS... ');
    Writeln; PrintTree (CONS (Root1,Root2),0);
    Writeln('----------------------------------------');
    Writeln
      ('Demonstration of the ATOM,CAR,CDR functions...');
    If  ATOM (Root1)
```

```
         then PrintTree (CAR (CONS (Root1,Root2)),0)
         else PrintTree (CDR (CONS (Root1,Root2)),0);
      Writeln('-----------------------------------');
      Writeln ('Demonstration actions functions EQUAL... ');
      If   EQUAL (Root1,Root2)
          then Writeln ('S-expressions are equal...')
          else Writeln ('S-expressions are not equal...');
      Writeln('-----------------------------------');
      Writeln('Demo actions functions EQ...');
      If    (ATOM (Root1)) AND (ATOM (Root2))
                           AND (EQ (Root1,Root2))
          then   Writeln ('Atoms are equal...')
          else   Writeln ('Atoms are not equal...');
      Writeln('--------------------------------------');
      Writeln ('Demonstration of finding the "leftmost" atom');
      Writeln (FIRSTATOM (CONS (Root1,Root2)));
      Writeln('-----------------------------------------');
      Writeln ('Demonstration actions functions VAL... ');
      If  ATOM (Root1)
         then Writeln (VAL (Root1))
         else Writeln
                ('The VAL function only applies to atoms...')
   END.
```

[(1)] Alagic S., Arbib M. Design of correct structured programs. - M.: Radio and Communications, 1984. - 264 p.
[(2)] Bauer F.L., Gooz G. Computer Science. Introductory Course: In 2 parts. Part 1. translated from German. - M.: Mir, 1990. - 336 p.; Part 2. translated from German. - M.: Mir, 1990. - 423 p.

In the next step we will look at ***the use of binary trees with labeled leaves***.

Step 103.
LISP and TURBO Pascal. Using Binary Trees with Labeled Leaves

In this step we will look at ***using binary trees with labeled leaves to encode information***.

Binary trees with labeled leaves can be used as code trees for ***Fano coding*** [1, part 1, p. 152].

It consists of replacing each element of a binary tree with a sequence of symbols **L** and **O**, where the length of this sequence is equal to the depth of the element in the tree.

Here is a function for encoding in **LISP** and **Pascal**:

```
   (DEFUN COD (LAMBDA (X LST)
    ; ------------------------------------------------ ;
   ; Fano coding: X is the encoded "leaf" of the tree, ;
   ; LST is a binary tree with labeled leaves ;
   ; ------------------------------------------------ ;
       (COND ( (ATOM LST) NIL )
             ( (CONTAINS X (CAR LST))
                       (CONS O (COD X (CAR LST))) )
             ( (CONTAINS X (CDR LST))
                       (CONS L (COD X (CDR LST))) )
       )
   ))
   ; --------------------------- ;
   (DEFUN CONTAINS (LAMBDA (X LST)
; Predicate for verifying the belongingness of the X list LST ;
    (COND ( (ATOM LST) (EQ X LST) )
          (  T  (OR (CONTAINS X (CAR LST))
                    (CONTAINS X (CDR LST))) )
    )
))
FUNCTION  Cod (S: Lisp; X: TipElement): BitString;
{ ---------------------------------------------- }
{  Fano coding                                 }
{  S - binary tree with marked leaves, }
{  X is the "content" of the S leaf               }
{ ---------------------------------------------- }
BEGIN
   If  ATOM (S)
      then  Cod := ''
      else  If  Contains (CAR(S),X)
               then  Cod := Concat ('O',Cod (CAR(S),X))
               else  If  Contains (CDR(S),X)
                        then  Cod := Concat ('L',Cod (CDR(S),X))
END;
{ ------------------------------------------------------- }
FUNCTION  Contains (S: Lisp; X: Char): Boolean;
BEGIN
   If  ATOM (S)
      then  Contains := X=VAL(S)
      else  Contains := Contains (CAR(S),X) OR
                               Contains (CDR(S),X)
END;
```

To ***decode*** the sequence of characters **O, L** we have the algorithm:

```
(DEFUN DECOD (LAMBDA (CODE LST)
 ; Decoding Fano: CODE - Fano code, ;
; LST - binary tree with marked leaves ;
    (COND ( (OR (ATOM LST)) ( NULL CODE)) LST )
          ( (EQ (CAR CODE) O)
                      (DECOD (CDR CODE) (CAR LST)) )
          ( (EQ (CAR CODE) L)
                      (DECOD (CDR CODE) (CDR LST)) )
    )
))
FUNCTION   Decode (S: Lisp; A: BitString): TipElement;
 { Fano Decoding }
{ A is a Fano code, }
{ S is a binary tree with labeled leaves }
BEGIN
   If   (ATOM (S)) OR (A='')
       then   Decode := VAL(S)
       else   If   Copy (A,1,1) = 'O'
                then   Decode :=
                          Decode (CAR(S),Copy (A,2,Length(A)-1))
               else  If  Copy (A,1,1) = 'L'
                        then   Decode  :=
                                  Decode (CDR(S),
                                             Copy (A,2,Length(A)-1))
END;
```

Example.

```
PROGRAM   CodeFano;      {$A-}
{ Fano encoding and decoding }
{ "leaves" of dotted pairs }
   type   TipElement = Char;
```

```
            Lisp       = ^LispEl;
            LispEl     = Record
                             Case Tag: (Single,Pair) of
                                        { А т о м  }
                                Single: (Leaf : TipElement);
                                        { Dotted pair }
                                Pair  : (Left : Lisp;
                                         Right: Lisp)
                             End;
            Pod = String [23];
            BitString  = String [50];
     { -------------------------------------------------- }
      var   Root : Lisp;       { Pointer to dotted pair }
           Strk : Stroka;     { String - dotted pair }
           Result: Profession;
           i : Integer;    { Helper variable }
           Code   : BitString;
           Symbol: TipElement;
  { --------------------------------- }
   PROCEDURE   Enter ( var T: Lisp);
    { Build a binary tree }
   { corresponding to a LISP S-expression }
      var   X: TipElement;
    BEGIN
      Write (' '); X := Strk[i]; i := i + 1;
     { Put the dotted pair element X into the binary tree }
      If   X='#'
          then   begin
                   New (T); T^.Tag := Pair;
                   Enter (T^.Left); Enter (T^.Right)
                 end
         else   begin
                   New (T); T^.Tag := Single;
                   T^.Leaf := X
                 end
   END ;
   { ------------------------------------------------- }
   PROCEDURE   PrintTree  (W: Lisp; l: Integer);
    { Output the binary tree corresponding to }
   { dotted pair W }
      var   i: Integer;
    BEGIN
      If   W^.Tag <> Single
          then   begin
                   PrintTree (W^.Right,l+1);
                   For i:=1 to l do  Write ('   ');
                   Writeln('#');
                   PrintTree (W^.Left,l+1)
                 than
         else   begin
                   For i:=1 to l do  Write ('   ');
                   Writeln (W^.Leaf)
                 end
   END ;
   { ------------------------------------- }
   PROCEDURE   Convert (Strk: Stroka;
                              var Result: Stroka);
       var   k : Integer;   { Parameter index }
           Ch: Char;       { Generalized symbol }
   BEGIN
      Result := '';
      For k:=1 to Length (Strk)  do
         begin
            Ch := Strk[k];
            If  Ch='('
              then  Result := Result + '#'
              else  If  (Ch<>')') AND (Ch<>'.')
                                  AND (Ch<>' ')
                     then  Result := Result + Ch
        end
  END;
 { -------------------------------- }
  FUNCTION  ATOM (X: Lisp): Boolean;
  { Checking the Argument Type }
  BEGIN
     ATOM := (X^.Tag = Single)
  END ;
  { ----------------------------- }
  FUNCTION   CAR (X: Lisp): Lisp;
   { Extract the first component of an S-expression }
  BEGIN
     CAR := X^.Left
  END ;
  { ----------------------------- }
  FUNCTION   CDR (X: Lisp): Lisp;
   { Extract the second component of the S-expression }
  BEGIN
     CDR := X^.Right
  END;
 { ---------------------------------- }
  FUNCTION  VAL (A: Lisp): TipElement;
  BEGIN
     VAL := A^.Leaf
  END;
 { ---------------------------------------------------- }
  FUNCTION  Contains (S: Lisp; X: Char): Boolean;
  BEGIN
     If  ATOM (S)
        then  Contains := X=VAL(S)
        else  Contains :=
                         Contains (CAR(S),X) OR
                         Contains (CDR(S),X)
  END ;
  { ------------------------------------------------- }
  FUNCTION   Cod (S: Lisp; X: TipElement): BitString;
   { Fano encoding: }
  { S is a binary tree with labeled leaves, }
  { X is the "contents" of a leaf of S }
  BEGIN
     If   ATOM (S)
         then   Cod := ''
         else   If   Contains (CAR(S),X)
                  then   Cod := Concat ('O',Cod (CAR(S),X))
                  else   If   Contains (CDR(S),X)
                          then   Cod :=
                                Concat('L',Cod (CDR(S),X))
  END ;
  { ------------------------------------------------------- }
  FUNCTION   Decode (S: Lisp; A: BitString): TipElement;
   { Fano Decoding: }
  { A is a Fano code, }
  { S is a binary tree with labeled leaves }
  BEGIN
     If   (ATOM (S)) OR (A='')
         then   Decode := VAL(S)
         else   If   Copy (A,1,1) = 'O'
                  then   Decode :=
                          Decode (CAR(S),
                                  Copy (A,2,Length(A)-1))
                 else  If  Copy (A,1,1) = 'L'
```

```
                            then   Decode  :=
                                     Decode (CDR(S),
                                     Copy (A,2,Length(A)-1))
   END;
  { ---- }
   BEGIN
      Writeln
     ('Let's construct a binary tree with labeled leaves');
      Writeln('given a Lisp dotted pair...');
      Writeln('Enter dotted pair...');
      i := 1; ReadLn (Strk); Convert (Strk,Result);
      Strk := Result; Root := Nil; Enter (Root);
      Writeln; PrintTree (Root,0); Writeln;
      Writeln('---------------------------------------');
      For i:=1 to 4  do
         begin
            Write ('Fano encoding of symbol ');
            ReadLn (Symbol);
            If  Contains (Root,Symbol)
               then  Writeln (' ... ',Cod (Root,Symbol))
               else  Writeln (' - There is no symbol in the tree!')
         end;
      Writeln('---------------------------------------');
      Writeln('Let's start decoding...');
      For i:=1 to 4  do
         begin
            Write ('Enter the code... '); ReadLn (Code);
            Write (' - ');
            If   Cod (Root,Decode (Root,Code)) = Code
                then   Writeln (Decode (Root,Code))
                else   Writeln ('The code is invalid!')
          end
   END .
```

---

(1) Bauer F.L., Goos G. Computer Science. Introductory Course: In 2 parts. Part 1. translated from German. - Moscow: Mir, 1990. - 336 p.; Part 2. translated from German. - Moscow: Mir, 1990. - 423 p.

In the next step we will get acquainted with ***arbitrary trees with labeled leaves*** .

Step 104.
Random trees with marked leaves

In this step we will provide ***general information about arbitrary trees with labeled leaves*** .

Now we can introduce the general notion ***of a tree with labeled leaves***. Binary trees with labeled leaves are a special case of them.

Let us accept the following definition:

A tree with labeled leaves is a sequence of branches, and a branch is

- or an object of the base type,
- or a tree with marked leaves.

Pascal **does** not provide a data type for "arbitrary trees with labeled leaves", but we can construct an encapsulated data type with this name.

To do this, first, in accordance with the recursive definition of an arbitrary tree with labeled leaves, we obtain the following description of the type [1, v.2, pp.393-394,439-443]:

```
type  Tree   = ^PlexEl;
      PlexEl =  Record
                   O: Tree;
                   Case Atomic: Boolean of
                     TRUE : (Atom: Char);
                     FALSE: (F    : Tree)
                End;
```

Please note that in **TURBO Pascal** there are limitations: a record can contain no more than one variant component, and this component must be the last one in the record.

The field names come from:

- **F - "First branch** ";
- **O - "Other branches** "

The constructor functions required for constructing an arbitrary tree with marked leaves are implemented as follows [1, v.2, pp.439-443]:

```
FUNCTION   EmptyTree : Tree;
 { Build an empty tree }
BEGIN
   EmptyTree := Nil
END ;
{ ---------------------------------------------- }
FUNCTION   Append (C: Char; B: Tree): Tree;
 { Add a "leaf" C to tree B }
   var   h: Tree;
 BEGIN
   New (h); h^.Atomic := TRUE; h^.Atom := C; h^.O := B;
   Append := h
END;
```

Let's graphically illustrate the action of the function **Append('C', B)** :

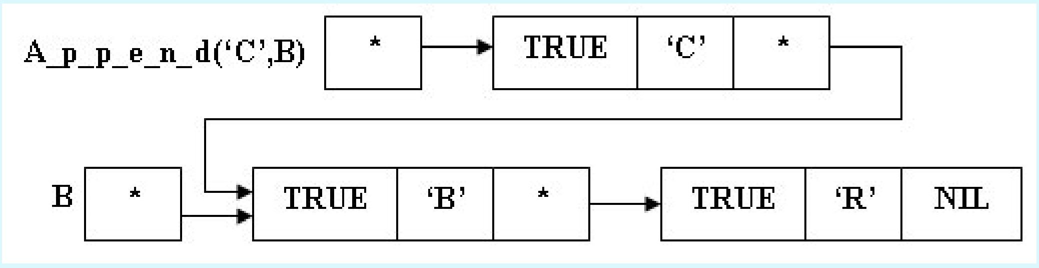


Fig.1. Result of the function **Append('C',B)**

```
FUNCTION   Join (A: Tree; B: Tree): Tree;
 { Join trees A and B to form a common root }
   var   h: Tree;
 BEGIN
   New (h); h^.Atomic := FALSE; h^.F := A; h^.O := B;
   Join    := h;
END;
```

Let us illustrate graphically the action of the function **Join (A, B)** :

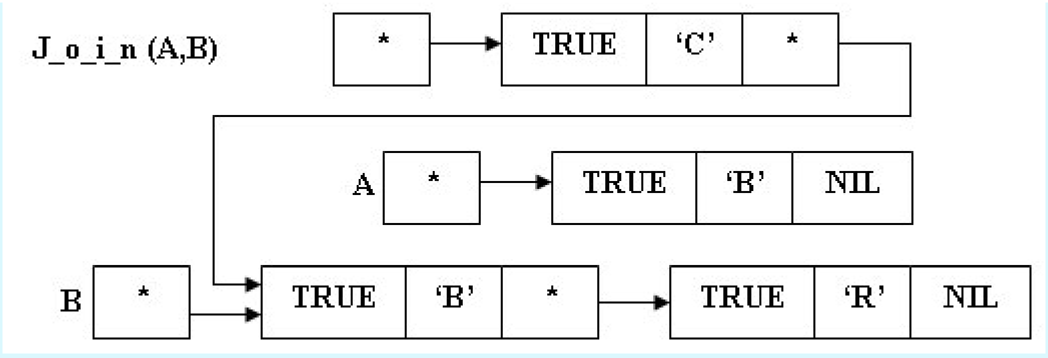


Fig.2. Result of the function **Join (A, B)**

Term for a tree with labeled leaves composed of objects **A, B, C, D, E, F**

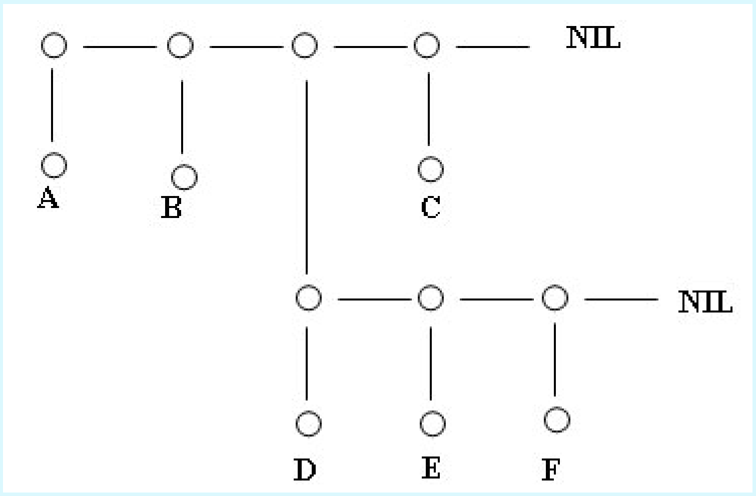


Fig.3. Term

has the form:

```
  L := Append (A,Append (B,
          Join (
                    Append (C,EmptyTree),
                    Append (D,Append (E,
                    Append (F,EmptyTree)))
                  )
       ))
```

Example 1.

```
  PROGRAM   LISPAS;      {$A-}
{ Build a term and display it on the display screen }
     type   TipElement = Char;
           Tree    = ^PlexEl;
           PlexEl =  Record
                        O: Tree;
                        Case Atomic: Boolean of
                           TRUE : (Atom: TipElement);
                           FALSE: (F   : Tree)
                     End ;
      var    Ukaz0,Ukaz1,Ukaz2,Ukaz3,Ukaz4,Ukaz5,Ukaz6: Tree;
           Root: Tree;
{ ------------------------------------------ }
  FUNCTION   Join (A: Tree; B: Tree): Tree;
   { Join trees A and B to form a common root }
     var   h: Tree;
```

```
    BEGIN
      New (h); h^.Atomic := FALSE; h^.F := A; h^.O := B;
      Join   := h;
   END ;
   { ---------------------------------------------------- }
   FUNCTION   Append (C: TipElement; B: Tree): Tree;
    { Add "leaf" C to tree B }
      var   h: Tree;
    BEGIN
      New (h); h^.Atomic := TRUE; h^.Atom := C; h^.O := B;
      Append := h
   END;
  { --------------------------------- }
   FUNCTION  EmptyTree : Tree;
   BEGIN
      EmptyTree := Nil
   END;
  { ----------------------------------------------------- }
   PROCEDURE  PrintTree (Root: Tree; l: Integer);
   { display the root binary tree on the display}
      var  i: Integer;
   BEGIN
      If  Root^.Atomic
         then  begin
                  If  Root^.O<>Nil
                     then  PrintTree (Root^.O,l)
                     else;
                  For i:=1 to l do  Write ('   ');
                  Writeln (Root^.Atom)
               end
         else  begin
                  If  Root^.O<>Nil
                     then  PrintTree (Root^.O,l)
                     else;
                  PrintTree (Root^.F,l+1);
                  For i:=1 to l do  Write ('   ');
                  Writeln('#')
               end
   END ;
   { --- }
   BEGIN
      Writeln('Let's build two trees with labeled leaves...');
      Writeln('-----------------------------------------');
      Ukaz0  := Append ('F',EmptyTree);
      Ukaz1  := Append ('E', Ukaz0);
      View2  := Append ('D',View1);
      Ukaz3  := Append ('K',EmptyTree);
      Ukaz4  := Append ('C',Ukaz3);
      Ukaz6  := Join  (Ukaz4, Nil );
      Ukaz5  := Join  (Ukaz2,Ukaz6);
      Ukaz1  := Append ('B',Ukaz5);
      Ukaz0  := Append ('A', Ukaz1);
      PrintTree (Ukaz0,0); Writeln;
      Writeln('-----------------------------------------');
      Ukaz0  := Append ('F',EmptyTree);
      Ukaz1  := Append ('E', Ukaz0);
      View2  := Append ('D',View1);
      Ukaz3  := Append ('K',EmptyTree);
      Ukaz4  := Append ('C',Ukaz3);
      Ukaz5  := Join  (Ukaz2,Ukaz4);
      Ukaz1  := Append ('B',Ukaz5);
      Ukaz0  := Append ('A', Ukaz1);
      PrintTree (Ukaz0,0); Writeln
   END.
```

Test example:

Let's build two trees with marked leaves...

```
--------------------------------------------------
   K
   C
#
   F
   AND
   D
#
B
A
--------------------------------------------------
K
C
   F
   AND
   D
#
B
A
```

The results obtained can be graphically represented as follows:

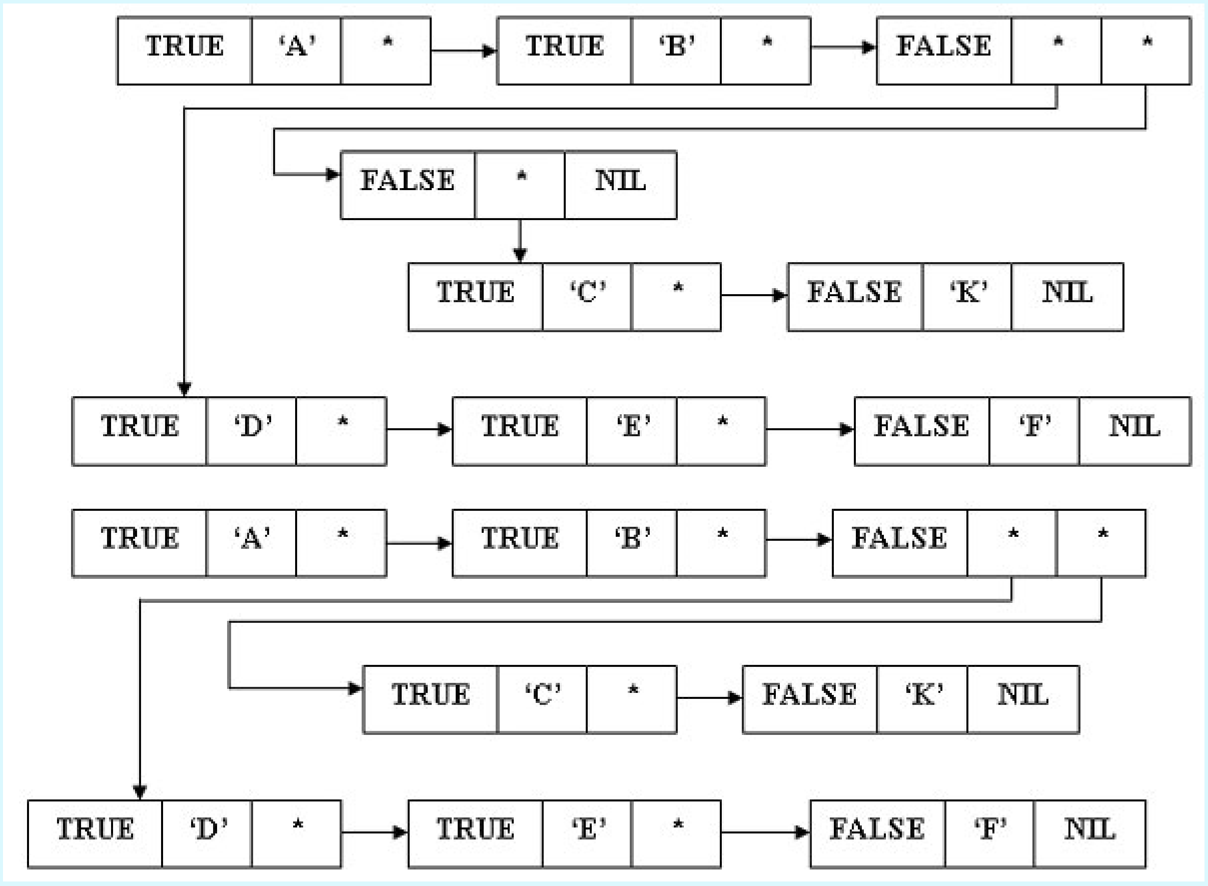


Fig.4. Graphic interpretation

The following program shows how a LISP list **entered** from the keyboard can be converted into a tree with labeled leaves.

Example 2.

```
 PROGRAM   LISPAS;     {$A-}
{ Representation of a LISP list as an arbitrary tree with }
{ marked leaves }
    type   TipElement = Char;
          Tree        = ^PlexEl;
          PlexEl      =  Record
                             O: Tree;
                             Case Atomic: Boolean of
                                TRUE : (Atom: TipElement);
                                FALSE: (F    : Tree)
                          End;
          Q           = String [23];
    var    Root: Tree; { The resulting tree with the }
                      { leaves marked }
          List: Q;     { The LISP input list }
          i    : Integer;
{ --------------------------------- }
 PROCEDURE   Enter ( var T: Tree);
  { Build a tree corresponding to a LISP list }
    var   X: TipElement;
  BEGIN
    X := List[i]; i := i + 1;
    If   X<>'.'
        then   begin
       { Place element X of Lisp list into tree }
                If   X='('
                    then   begin
                             New (T); T^.Atomic := FALSE;
                             Enter (T^.F);
                             Enter (T^.O)
                          end
                   else  If  (X IN ['A'..'z'])
                             then  begin
                                      New (T);
                                      T^.Atomic := TRUE;
                                      T^.Atom := X;
                                      Enter (T^.O)
                                   end
                             else   If   X=')'
                                       then   T := Nil
             end
 END ;
 { ------------------------------------------------- --- }
 PROCEDURE   PrintTree (Root: Tree; l: Integer);
  { Output of an arbitrary tree with labeled leaves }
    var   i: Integer;
  BEGIN
    If   Root^.Atomic
        then   begin
                If   Root^.O<> Nil
                   then   PrintTree (Root^.O,l)
                    else ;
                For i:=1 to l do  Write ('   ');
                Writeln (Root^.Atom)
             end
       else  begin
                If  Root^.O<>Nil
                   then  PrintTree (Root^.O,l)
                   else;
                PrintTree (Root^.F,l+1);
                For i:=1 to l do  Write ('   ');
                Writeln('#')
             end
 END ;
 { --- }
 BEGIN
    Writeln('Build a tree with labeled leaves');
    Writeln('given a Lisp list...');
    Writeln
    ('Enter a LISP list without spaces between atoms.');
    Writeln
    ('The last character must be the "."...' character);
    Read (List);
    i := 1; Root := Nil;
    Enter (Root); Writeln;
    PrintTree (Root,0); Writeln
 END.
```

Test example:

```
 Let's build a tree with marked leaves
 given a Lisp list...
 Enter the Lisp list...
 (CU((KE)N)(G(I(T)))).
             T
          #
          I
       #
       G
    #
       N
          AND
          K
       #
    #
    IN
    C
 #
```

Notes .

1. *Let's consider the implementation of the remaining operations on arbitrary trees with marked leaves:*

```
 FUNCTION  IsEmptyTree  (A: Tree): Boolean;
 BEGIN
    IsEmptyTree := A=Nil
 END;
{ ---------------------------------------- }
 FUNCTION  IsAtom (A: Tree): Boolean;
 { NOT IsEmptyTree (Tree) }
 BEGIN
    IsAtom := A^.Atomic
       { AND  IsEmptyTree (A^.O) }
 END;
{ ----------------------------------------------- }
 FUNCTION  FirstBranch (A: Tree): Tree;
    { NOT IsEmptyTree (A) AND
      NOT IsAtom (A) }
 BEGIN
    FirstBranch := A^.F
 END;
{ ---------------------------------------------------- }
 FUNCTION  OtherBranches (A: Tree): Tree;
 { NOT IsEmptyTree (A) }
 BEGIN
```

```
      OtherBranches := A^.O
   END;
 { --------------------------- }
   FUNCTION  VAL (A: Tree): Char;
   { IsAtom (a) }
   BEGIN
      VAL := A^.Atom
   END;
```

2. *Trees with labeled leaves serve primarily to represent homogeneous hierarchical structures and those structures in which the degree of branching varies from vertex to vertex even within one "level". However, with such a representation the "color" of the representation is lost. That is why arbitrarily branching labeled trees, in the vertices of which additional information ("attributes") can be placed, are of great practical importance in system programming.*

---

(1) Bauer F.L., Goos G. Computer Science. Introductory Course: In 2 parts. Part 1. translated from German. - Moscow: Mir, 1990. - 336 p.; Part 2. translated from German. - Moscow: Mir, 1990. - 423 p.

In the next step we will begin to look at the relationship between **LISP *and* LOGO *languages***.

Step 105.
LISP and LOGO

In this step we will look at the relationship between **the LISP *and* LOGO *programming languages***.

Let's try to answer the question: why did S. Papert create a language ***based on the* LISP** language for teaching children?

Here are some quotes from the monograph [1]: “LOGO is the designation of the philosophy of learning with the help of an expanding family of programming languages, which this philosophy gave birth to. The characteristic features of the LOGO family of languages include ***functionality*** and ***recursion***. Thus, in LOGO it is possible to introduce new commands and functions, which can then be used in exactly the same way as elementary commands and functions.

LOGO is a language that allows ***interpretation***. This means that LOGO can be used in a dialog mode. Modern programming systems in the LOGO language are integral list structures, i.e. they implement list languages, which include lists themselves, lists of lists, etc.

An example of the fruitful use of the list structure is the representation of LOGO procedures as ***a list of lists***, which allows these procedures to be created, modified, and included in other LOGO procedures. Thus, LOGO is not a toy, but a real programming language, but only for children...”[p.17-18].

“In 1968-69, for the first time, a group of 12 average seventh-graders from a mixed school for young people in Lexington, Massachusetts, worked with LOGO instead of the regular school mathematics program for an entire school year. At that time, the LOGO system did not have a graphical version. The students wrote programs that translated English into dialects used by Hispanic minorities living in the United States, programs that played strategy games, programs for writing poetry. The work of these children was the first confirmation that the LOGO language was suitable for teaching on computers to children for whom these devices were new. However, I wanted to make sure that this language was suitable for teaching fifth-graders, third-graders, and even preschoolers. It was also clear to me that if LOGO could be used for teaching at these ages, then the work on thematic programming should be excluded” [p. 21].

“The name LOGO was chosen for the new language in order to emphasize that this language is first and foremost ***symbolic*** and only then quantitative” [p.208].

We hope that you will be able to draw your own conclusion about the strong similarity between the **LOGO** and **LISP** programming languages after viewing a number of examples of solving school programming problems in LOGO and LISP. In the future, we plan to post material on the **LOGO** language on the site.

---

*Note : All **LOGO** examples are developed for **LOGOWriter** version 3.1.*

Example 1. Counting the number of **"A"** characters in a given word **LST**.

a) Program in **LISP** language:

```
(DEFUN COUNT (LAMBDA (WORD)
   (COUNT_LETTER (UNPACK WORD))
))
; ------------------------------ ;
(DEFUN COUNT_LETTER (LAMBDA (LST)
   (COND ( (NULL LST) 0)
         ( (EQ (CAR LST) A)
              (+ 1 (COUNT_LETTER (CDR LST))) )
         (   T   (COUNT_LETTER (CDR LST)) )
   )
))
```

b) Program in **LOGO** language (functional programming style):

```
THIS IS A COUNT :SL
    IFELSE EMPTY? :SL [ OUTPUT 0]
    [
      IFELSE EQUAL? FIRST: SL "A
      [ OUTPUT 1 + (COUNT KPRV :SL)]
      [ OUTPUT COUNT KPRV :SL ]
    ]
END
```

Test example:

```
SHOW THE COUNT "ADAABC
3
```

Example 2. Crossing out all the letters **"A"** from the given word **LST**.

a) Program in **LISP** language:

```
(DEFUN DELETE (LAMBDA (WORD)
   (DELETE_LETTER (UNPACK WORD))
))
; -------------------------------- ;
(DEFUN DELETE_LETTER (LAMBDA (LST)
   (COND ( (NULL LST) NIL)
         ( (EQ (CAR LST) A) (DELETE_LETTER (CDR LST)) )
         (  T  (CONS (CAR LST) (DELETE_LETTER (CDR LST))) )
   )
))
```

b) Program in **LOGO** language (functional programming style):

```
THIS IS A CROSS-OUT :SL
    IFELSE EMPTY? :SL [ EXIT "]
    [
      IFELSE EQUAL? FIRST: SL "A
      [ EXIT CROSSING OUT KPRV :SL ]
      [ EXIT (WORD PRV :SL CROSSING OUT KPRV :SL) ]
    ]
END
```

Test example:

```
  SHOW CROSSOUT "ADAABC
  DBC
```

Example 3. Replacing all the letters **"A"** from the word **LST** with the letter **"B"**.

a) Program in **LISP** language:

```
   (DEFUN REPLACE (LAMBDA (WORD)
      (REPLACE_A_B (UNPACK WORD))
   ))
   ; ---------------------------- ;
   (DEFUN REPLACE_A_B (LAMBDA (LST)
      (COND ( (NULL LST) NIL)
            ( (EQ (CAR LST) A) (CONS B (REPLACE_A_B (CDR LST))) )
            (  T  (CONS (CAR LST) (REPLACE_A_B (CDR LST))) )
      )
   ))
```

b) Program in **LOGO** language (functional programming style):

```
THIS IS A REPLACEMENT: SL
    IFELSE EMPTY? :SL [ EXIT "]
    [
      IF ELSE EQUAL? FIRST: SL "A
       [EXIT (WORD "B REPLACEMENT KPRV:SL)]
       [EXIT (WORD PRV:SL REPLACEMENT KPRV:SL)]
    ]
TIME
```

Test example:

```
  SHOW REPLACEMENT "ADAABC
  BDBBBC
```

Example 4. Reversing a given list **LST**.

a) Program in **LISP** language:

```
   (DEFUN REVERSE1 (LAMBDA (LST)
    ; The function reverses the list LST at the very first level ;
      (COND ( (NULL LST) NIL )
            (  T  (APPEND (REVERSE1 (CDR LST)) (LIST (CAR LST))) )
      )
   ))
   ; ----------------------- ;
   (DEFUN APPEND (LAMBDA (X Y)
   ; The APPEND function returns a list consisting of the elements of ;
   ; list X appended to list Y ;
      (COND ( (NULL X) Y )
            ( T (CONS (CAR X) (APPEND (CDR X) Y)) )
      )
   ))
```

b) Program in **LOGO** language (functional programming style):

```
THIS IS AN APPEAL: SL
    IFELSE EMPTY? :SL [ OUTPUT :SL]
    [EXIT (WORD OF ADDRESS KPRV :SL FIRST :SL)]
TIME
```

Test example:

```
  SHOW APPEAL [1 2 3]
  [3 2 1]
```

Example 5. Determine the largest element of a single-level numeric list **LST**.

a) Program in **LISP** language:

```
   (DEFUN MAX (LAMBDA (LST)
      (COND ( (NULL LST) NIL )
            ( (EQ (LENGTH LST) 1)  (CAR LST) )
            (   T  (MAX2 (CAR LST) (MAXIM (CDR LST))) )
      )
   ))
   ; --------------------- ;
   (DEFUN MAX2 (LAMBDA (X Y)
      (( (> X Y) X )
                 AND )
   ))
```

b) Program in **LOGO** language (functional programming style):

```
THIS IS MAXIM :LST
    IFELSE EMPTY? :LST [ EXIT "]
    [
      IFELSE EQUAL? HOW MANY :LST 1
        [ EXIT ONE :LST]
        [ MAX2 OUTPUT FIRST :LST MAXIM KPRV :LST]
    ]
END
THIS IS MAX2 :X :Y
    IFELSE :X > :Y [ OUTPUT :X ]
    [ EXIT :Y ]
END
```

Test example:

```
SHOW MAX [1 3 2]
3
```

Example 6. Determine whether the elements of a list of integers are increasing.

a) Program in **LISP** language:

```
 (DEFUN ERASEP (LAMBDA (LST)
    (COND ( (NULL LST) T )
          ( (EQ (LENGTH LST) 1) T )
          ( (< (CAR LST) (CADR LST))
              (ERASEP (CDR LST)) )
          (  T  NIL )
    )
 ))
```

b) Program in **LOGO** language (functional programming style):

```
THIS IS MON :SP
    IFELSE EMPTY? :SP [ OUTPUT "TRUE" ]
      [ IFELSE EQUAL? (HOW MANY :SP) 1 [ OUTPUT "TRUE ]
         [IFELSE(FIRST:SP) < (FIRST KPRV:SP)
             [ EXIT MON KPRV :SP ]
             [ EXIT "FALSE ]
         ]
      ]
END
```

Test examples:

```
SHOW MON [1 3 2]
LIE
SHOW MON [1 2 3]
THE TRUTH
```

Example 7. Sorting a numeric list in ascending order using the ***simple exchange*** method.

b) Program in **LOGO** language (functional programming style):

```
ETO PRIMEP:LST
   IF ORDERED :LST
      [GOODBYE :LST]
      [GIVE AN EXAMPLE OF SORT :LST ]
TIME
THIS IS TYPE:LST
   IF EMPTY? :LST
      [ISSUE [] ]
      [IF EQUAL? HOW MANY :LST 1
          [GOODBYE :LST]
          [IF (FIRST :LST) > (FIRST WITHOUTFIRST :LST)
              [GIVE OUT COMBINE
                       (FIRST WITHOUT FIRST :LST)
                       (SORT UNION (FIRST :LST)
                                    (THE FIRST OF THE FIRST :LST ))]
              [GIVE OUT UNITE (FIRST :LST)
                              (SORT PRIMELESS :LST)]
          ]
      ]
TIME
THIS IS ORDERED:LST
   IF EMPTY? :LST
      [GIVE "TRUE"]
      [IF EQUAL? (HOW MANY :LST) 1
          [GIVE "TRUE"]
          [IF (FIRST :LST) > (FIRST WITHOUTFIRST :LST)
              [GIVE "LIE"]
              [ISSUE ORDERLY BEZFIRST :LST]
          ]
      ]
TIME
```

Example 8. Sorting a numeric list in ascending order using the ***linear insertion*** method [2].

b) Program in **LOGO** language (functional programming style):

- procedure ***SORTING*** - sorting an unordered list **L** using the linear insertion method;
- procedure ***INSERT*** adds an element **A** to an ordered list **L** in such a way that the order is preserved;
- The procedure ***INSERT*** allows the elements of the list **NEW** to be inserted into the ordered list **L**.

```
THIS IS A SORTING :L
   IF ELSE EMPTY? :L
    [ EXIT :L ]
    [ EXIT INSERT
              FIRST :L
              SORTING KPRV:L
    ]
END
THIS IS AN INSERTION :A :L
   IF ELSE EMPTY? :L
     [ VNSP EXIT :A :L]
     [ IF ELSE :A < FIRST :L
         [ VNSP EXIT :A :L]
         [ OUTPUT VNSP FIRST :L INSERT :A KPRV :L ]
     ]
END
THIS IS A BREAKUP :NEW :L
   IF ELSE EMPTY? :NEW
     [ EXIT :L]
     [INSERT ONE :NEW PARTING KPRV :NEW :L ]
END
```

(1) Papert S. Revolution in consciousness: children, computers and fruitful ideas: Trans. from English / Ed. by A. V. Belyaeva, V. V. Leonas. - M.: Pedagogy, 1989. - 224 p.
(2) Lorin G. Sorting and sorting systems. - M.: Science, 1983. - 384 p.

In the next step we will look at the relationship between **LISP *and* PROLOG *languages***.

Step 106.
LISP and PROLOG

In this step we will look at the relationship between **the LISP *and* PROLOG *programming languages***.

Functional definitions are easily translated into relational notations if we use [see 1, p. 147]:

- ***equivalence***

```
f(x) <--> F(x y),              (E1)
```

where **F** is a predicate symbol corresponding to the function symbol **f**, and

- ***properties of the equality relation***

```
P(f(x)) <--> For-all y [ P(y) if f(x)=y ] (E2)
P(f(x)) <--> There is y [ P(y) and f(x)=y ] (E3)
```

Here **P(t)** represents any expression containing the term **t** . The symbol **"<-->"** (read "if and only if") expresses equivalence. ***For-all*** and ***Exists*** are the universal quantifier and the existential quantifier, respectively.

Let us give a number of examples of translating programs written in **LISP** into programs in **Prolog**.

Example 1. Here is a LISP program **that** copies a list of integers **LST**:

```
(DEFUN COPY (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         (  T  (CONS (CAR LST) (COPY (CDR LST))) ))))
```

Let's use "more equational" notations:

```
COPY (NIL) = NIL
COPY (LST) = (CONS CAR (LST) COPY (CDR (LST)))
```

Now let's apply "TURBO-Prolog" notations:

```
COPY (NIL) = NIL
COPY ([X|Y]) = [X | COPY (Y)]
```

Applying (E1-E3) to the definitions **of COPY**, we obtain sequentially:

```
COPY (NIL) = NIL <--> copy ([],[]).
COPY ([X|Y]) = [X | COPY (Y)] <-->
copy ([X|Y],[X|COPY (Y)] <-->
copy ([X|Y],[X|Z]), if Z = COPY (Y) <-->
copy ([X|Y],[X|Z]), if copy (Y,Z) <-->
copy ([X|Y],[X|Z])  :- copy (Y,Z).
```

Each translation step consists of applying equivalence to replace a subexpression with an equivalent subexpression.

And finally, we change the names:

```
copy_list ([],[]).
copy_list ([Head|Tail],[Head|Tail1]) :-
                     copy_list (Tail,Tail1).
```

Example 2. Here is a LISP program **that** calculates the number of elements in an integer list **LST**:

```
(DEFUN LENGTH (LAMBDA (LST)
      (COND ( (NULL LST) 0 )
            (  T  (+ 1 (LENGTH (CDR LST)) ))))
```

Let's use "more equational" notations:

```
 LENGTH (NIL) = 0
 LENGTH (LST) = 1 + LENGTH (CDR LST)
```

Now let's apply "TURBO-Prolog" notations:

```
 LENGTH (NIL) = 0
 LENGTH ([X|Y]) = 1 + LENGTH (Y)
```

Applying (E1-E3) to the definitions **of COPY**, we obtain sequentially:

```
 LENGTH (NIL) = 0 <--> length ([],0).
 LENGTH ([X|Y]) = 1 + LENGTH (Y) <-->
 length ([X|Y],1+LENGTH (Y)) <-->
 length ([X|Y],Z), if (U=LENGTH (Y))&(Z=1+U) <-->
 length ([X|Y],Z), if (length (Y,U))&(Z=1+U) <-->
 length ([X|Y],Z):- length (Y,U), Z = 1 + U.
```

Each translation step consists of applying equivalence to replace a subexpression with an equivalent subexpression.

And finally, we change the names:

```
 sum_list ([],0).
 sum_list ([H|T],Sum) :- sum_list (T,Sum1), Sum = 1 + Sum1.
```

Example 3. Here is a **LISP** program that determines **the N-th** element of an integer list:

```
 (DEFUN NTH (LAMBDA (N LST)
    (COND ( (> N (LENGTH LST))   100 )
          ( (OR (< N 0) (ZEROP N)) -100 )
          ( (EQ N 1) (CAR LST) )
          (  T  (NTH (- N 1) (CDR LST)) ))))
```

Let's use "more equational" notations:

```
 NTH (N,LST) =  100, if N > LENGTH (LST)
 NTH (N,LST) = -100, if N <=0
 NTH (1,LST) =  CAR (LST)
 NTH (N,LST) =  NTH (N-1,CDR(LST))
```

Now let's apply "TURBO-Prolog" notations:

```
 NTH (N,LST)   =  100, if N > LENGTH (LST)
 NTH (N,LST) = -100, if N <=0
 NTH (1,[X|Y]) =  X
 NTH (N,[X|Y]) =  NTH (N-1,Y)
```

Applying (E1-E3) to the definitions **of COPY**, we obtain sequentially:

```
 NTH (N,LST) = 100, if N > LENGTH (LST) <-->
 nth (N,LST,100), if (Z=LENGTH (LST)) & (N>Z) <-->
 nth (N,LST,100), if (length (LST,Z)) & (N>Z) <-->
 nth (N,LST,100) :- length (LST,Z), N>Z.
```

```
   NTH (N,LST) = -100, if N <=0  <-->
   nth (N,LST,-100), if N <=0  <-->
   nth (N,LST,-100) :- N <=0.

   NTH (1,[X|Y]) = X <--> nth (1,[X|Y],X).
   NTH (N,[X|Y]) = NTH (N-1,Y) <-->
   nth (N,[X|Y],Z), if NTH (N-1,Y))=Z <-->
   nth (N,[X|Y],Z), if (U=N-1) & (nth (N-1,Y,Z)) <-->
   nth (N,[X|Y],Z) :- U=N-1, nth (U,Y,Z).
```

Each translation step consists of applying equivalence to replace a subexpression with an equivalent subexpression.

And finally, we change the names:

```
   find_n (N,LST,100):- sum_list (LST,V),N>V.
   find_n (N,LST,-100):- N<=0.
   find_n (1,[Head|_],Head).
   find_n (Middle,[_|Tail],Z) :-
                          Middle1 = Middle - 1,
                          find_n (Middle1,Tail,Z).
```

Example 4. Here is a LISP program **that** removes an element **X** from an integer list **LST**:

```
   (DEFUN DELETE (LAMBDA (X LST)
      (COND ( (NULL LST) NIL )
            ( (EQ (CAR LST) X) (DELETE X (CDR LST)) )
            (  T  (CONS (CAR LST) (DELETE X (CDR LST))) ))))
```

Let's use "more equational" notations:

```
   DELETE (NIL) = NIL
   DELETE (CAR(LST),LST) = DELETE (X,CDR(LST))
   DELETE (X,LST) = CONS (CAR(LST),DELETE (X,CDR(LST)))
```

Now let's apply "TURBO-Prolog" notations:

```
   DELETE (NIL) = NIL
   DELETE (X,[X|Y]) = DELETE (X,Y)
   DELETE (X,[Z|T]) = Z | DELETE (X,T)
```

Applying (E1-E3) to the definitions **of COPY**, we obtain sequentially:

```
   DELETE (NIL) = NIL <--> delete ([],[]).

   DELETE (X,[X|Y]) = DELETE (X,Y) <-->
   delete (X,[X|Y],Z), if DELETE (X,Y)=Z <-->
   delete (X,[X|Y],Z), if delete (X,Y,Z) <-->
   delete (X,[X|Y],Z) :- delete (X,Y,Z).

   DELETE (X,[Z|T]) = Z | DELETE (X,T) <-->
   delete (X,[Z|T],[Z|V]), if DELETE (X,T) = V <-->
   delete (X,[Z|T],[Z|V]), if delete (X,T,V) <-->
   delete (X,[Z|T],[Z|V]) :- delete (X,T,V).
```

Each translation step consists of applying equivalence to replace a subexpression with an equivalent subexpression.

And finally, we change the names:

```
    delete (_,[],[]).
    delete (X,[X|Tail],List) :-
                         delete (X,Tail,List).
    delete (X,[Z|Tail],[Z|V]):-
                         delete (X,Tail,V).
```

Program in **Prolog** language (**TURBO Prolog**, version 2.0).

```
 domains
    number = integer
    list   = integer *
 predicates
    read_list (list)
    write_list (list)
    copy_list (list,list)
    sum_list (list,number)
    find_n (number,list,number)
    append (list,list,list)
    reverse (list,list)
    trance (list,list,list)
    delete (number,list,list)
    listmax (list,number)
    greater (number,number,number)
 goal
    write ("Enter list items...")
    read_list (Lst), nl,
    write("Input result..."), write_list(Lst), nl,
    /* ------------------------------------------ */
    write ("Copy of the list... "), copy_list (Lst,Lst1),
    write_list (Lst1), nl,
    /* -------------------------------------- */
    write ("Number of elements in list...")
    sum_list (Lst,X1), write (X1), nl,
    /* ---------------------------------------------- */
    write ("Find an element with the given number in the list."),
    write("Enter number: "),
    readint (N), find_n (N,Lst1,T), write (T), nl,
    /* ---------------------------------------- */
    write("Let's merge two lists..."),
    append (Lst,Lst1,Lst2), write_list (Lst2), nl,
    /* ---------------------------------------- */
    write ("Reverse the given list."),
    write ("Enter list items...")
    read_list (Lst3), nl, reverse (Lst3,Lst4),
    write_list (Lst4), nl,
    /* --------------------------------------- */
    write ("Remove the specified element from the list."),
    write("Enter element...")
    readint (U), delete (U,Lst4,Lst5),
    write_list (Lst5), nl,
    write ("Find the largest element in a numeric list.")
    listmax (Lst5,A), write (A), nl.
 clauses
   /* ------------------------------------ */
   /* Entering a list of integers from the keyboard */
   /* ------------------------------------ */
    read_list ([H|T]):-
                         readint (H), !, read_list (T).
    read_list ([]).
   /* ----------------------------------------- */
   /* Output a list of integers to the display screen */
   /* ----------------------------------------- */
    write_list ([]).
    write_list ([Head|Tail]) :-
                         write (Head), write (" "),
                         write_list (Tail).
   /* ------------------------------ */
   /* Copying a list of integers */
   /* ------------------------------ */
    copy_list ([],[]).
    copy_list ([Head|Tail],[Head|Tail1]) :-
                         copy_list (Tail,Tail1).
   /* (DEFUN COPY (LAMBDA (LST) */
   /* (COND ( (NULL LST) NIL ) */
   /* ( T (CONS (CAR LST) (COPY (CDR LST))) ) */
   /* ) */
   /* )) */
   /* ---------------------------------- */
   /* Determine the number of elements in a */
   /* list of integers */
   /* ---------------------------------- */
    sum_list ([],0).
    sum_list ([_|T],Sum) :-
                         sum_list (T,Sum1),
                         Sum = 1 + Sum1.
   /* (DEFUN LENGTH (LAMBDA (LST) */
   /* (COND ( (NULL LST) 0 ) */
   /* ( T (+ 1 (LENGTH (CDR LST))) )))) */
   /* -------------------------------------- */
   /* Finding the list element with number N */
   /* -------------------------------------- */
    find_n (N,LST,100):-
                         sum_list (LST,V),N>V.
    find_n (N,_,-100):-
                         N<=0.
    find_n (1,[Head|_],Head).
    find_n (Middle,[_|Tail],Z) :-
                         Middle1 = Middle - 1,
                         find_n (Middle1,Tail,Z).
   /* (DEFUN NTH (LAMBDA (N LST)                    */
   /*    (COND ( (> N (LENGTH LST))   100 )      */
   /*          ( (OR (< N 0) (ZEROP N)) -100 )   */
   /*          ( (EQ N 1) (CAR LST) )               */
   /*          (  T  (NTH (- N 1)                   */
   /*                     (CDR LST)) ))))           */
   /* ------------------------- */
   /* "glueing" two lists */
   /* ------------------------- */
    append ([],L,L).
    append ([N|L1],L2,[N|L3]) :-
                         append(L1,L2,L3).
   /* ------------------------------------- */
   /* Remove a given element from the list */
   /* ------------------------------------- */
    delete (_,[],[]).
    delete (X,[X|Tail],List) :-
                         delete (X,Tail,List).
    delete (X,[Z|Tail],[Z|V]):-
                         delete (X,Tail,V).
   /* (DEFUN DELETE (LAMBDA (X LST)                    */
   /*    (COND ( (NULL LST) NIL )                      */
   /*          ( (EQ (CAR LST) X)                      */
   /*                      (DELETE X (CDR LST)) )      */
   /*          (  T  (CONS (CAR LST)                   */
   /*                      (DELETE X (CDR LST))) )))) */
   /* --------------------------- */
   /* "inversion" of list items */
   /* --------------------------- */
    reverse (L,Z)              :-
                         trance (L,[],Z).
```

```
      trance ([],Result,Result).
      trance ([X|Y],Result,Z)  :-
                           trance (Y,[X|Result],Z).
     /* -------------------------------------------- */
     /* Finding the largest element in a numeric list */
     /* -------------------------------------------- */
      listmax ([X|[]],X).
      listmax ([X|Y],Z) :-
                           listmax (Y,V), greater (X,V,Z).
      greater (X,Y,X)   :- X>Y.
      greater (X,Y,Y)   :- X<=Y.
   /* End of program */
```

Notes.

1. *Although functional notations are more familiar to the user than relational notations, computations via rewriting rules **are less** universal than backward reasoning.*

   For example, while a functional program can only be used to determine the length of a list, a relational program is reversible and can be queried like a database.

2. *This translation method gives a simple proof that Horn clauses can be used to represent all computable functions (this was first proved **by R. Hill** (see reference in [1, p.147]).*

   ***Sketch of proof**: Using the translation method illustrated above, any **system of Herbrand-Gödel equations** (representing a computable function) can be translated into an equivalent set of Horn clauses. Moreover, the effect of using an equation as a rewriting rule to replace terms that are compatible with the left-hand sides of equations with the right-hand sides of these equations can be modeled by using the corresponding clauses in the opposite direction - to reduce problems to subproblems. Such modeling associates with any computation by equations a structurally similar computation on Horn clauses that leads to the same result. This shows that **logic programs on Horn clauses** compute all functions that are computable by **the Herbrand-Gödel equations**, as desired.*

---

(1) Formal logic. - L.: Leningrad State University Publishing House, 1977. - 358 p.

In the next step we will consider ***elements of computational number theory***.

Step 107.
Elements of Computational Number Theory

In this step we will look at ***using* LISP *to solve computational problems*** .

Many theorems in number theory were originally discovered inductively by means of practical calculations, and only subsequently proved strictly mathematically... We must not forget, however, that results obtained inductively, even if they are based on a large number of experiments, cannot yet be considered firmly established. In number theory their validity is much weaker than in the experimental sciences.

The examples below show how one can test some number-theoretical conjectures related to "school" number theory.

Example 1.

```
; -------------------------------- ;
; Is the number 7^77 + 1 divisible by 5? ;
; -------------------------------- ;
(DEFUN P (LAMBDA (X A)
   (CAR (DIVIDE (SETQ S (+ (STEPEN X A) 1)) 5) )
   (COND
      ( (ZEROP (MOD S 5)) T )
      ( T NIL )
   )
))
; ----------------------- ;
(DEFUN STEPEN (LAMBDA (X A)
; Raising an integer X to a non-negative integer ;
; power of A ;
   (COND ( (ZEROP A) 1 )
         ( (ZEROP (- A 1)) X )
         (  T  (* (STEPEN X (- A 1)) X) )
   )
))
```

Example 2.

```
; ------------------------------------------- ;
; Make a list consisting of prime numbers ;
; segment [2,N] ;
; ------------------------------------------ ;
(DEFUN PAL (LAMBDA (N)
   (COND ( (EQ N 2) (CONS 2 NIL) )
         ( (ISPRIM N) (CONS N (PAL (- N 1))) )
         (    T    (PAL (- N 1)) )
   )
))
; Checking whether a number N is prime or not ;
(DEFUN ISPRIM (LAMBDA (N)
   (COND ( (EQ N 1) NIL )
         ( T (ISPR N 2) )
   )
))
; --------------------- ;
(DEFUN ISPR (LAMBDA (N M)
   (COND ( (EQ M N)                 T  )
         ( (EQ (MOD N M) 0) NIL )
         ( T (ISPR N (+ M 1)) )
   )
))
```

Example 3.

```
; ------------------------------------------------ ;
; Program for finding several palindromic ;
; numbers less than 10001 ;
; ------------------------------------------------ ;
(DEFUN PAL (LAMBDA ()
   (SETQ N 1)
   (SETQ LST NIL)
   (LOOP
       ( (EQ N 10001) (PRIN1 "Not Found!") )
       (COND ( (EQUAL (UNPACK N) (REVERSE (UNPACK N)))
                            (PRINT N) )
             ( T )
       )
       (SETQ N (+N 1))
   )
))
; ------------------- ;
(DEFUN PAL1 (LAMBDA (N)
   (COND ( (EQ N 1) (CONS 1 NIL) )
         ( (EQUAL (UNPACK N) (REVERSE (UNPACK N)))
                         (CONS N (PAL1 (- N 1))) )
         (   T   (PAL1 (- N 1)) )
   )
))
```

Example 4.

```
; ----------------------------------------------------- ;
; Determine whether a given natural number can be represented as ;
; the sum of the squares of two natural numbers. ;
; ----------------------------------------------------- ;
(DEFUN SUM (LAMBDA (N)
   (SETQ FIND NIL)
   (SETQ I1 1)
   (LOOP
       ( (> I1 N) )
       (SETQ I2 1)
       (LOOP
           ( (> I2 N) )
           ( (EQ N (+ (* I1 I1)
                      (* I2 I2))
             )
                  (SETQ FIND T) (SETQ A I1) (SETQ B I2)
           )
           (SETQ I2 (+ I2 1))
       )
       (SETQ I1 (+ I1 1))
   )
   (COND (FIND (PRIN1 A) (PRIN1 " ") (PRINT B) )
         (  T  (PRINT NO) )
   )
))
```

Example 5.

```
; --------------------------------------------------- ;
; Write a program to search among natural numbers ;
; n, n+1,..., 2n for so-called twins, i.e. two ;
; prime numbers whose difference is 2. If twins are ;
; found in the specified range, then return 1, ;
; if there are no twins, then return 0 ;
; --------------------------------------------------- ;
(DEFUN BLIZNECI (LAMBDA (X)
    (SETQ MAX (* 2 X))
    ( LOOP
          ( (> X MAX) )
          (SETQ Y (+ X 2))
          (COND ( (AND (SIMPLE X) (SIMPLE Y))
                      ((PRIN1 1) (PRIN1 " ") (PRIN1 X)
                       (PRIN1 " ") (PRINT Y))
                 )
                 (  T  0 )
          )
          (SETQ X (+ X 1))
    )
))
; --------------------- ;
(DEFUN SIMPLE (LAMBDA (N)
    (SETQ FLAG 1)
    (COND ( (AND (EQ (MOD N 2) 0) (NOT (EQ N 2))) NIL )
          (  T  ( (SETQ J 3)
                  (LOOP
                       ( (< (CAR (DIVIDE N 2)) J) )
                       ( (EQ (MOD N J) 0)
                             (SETQ FLAG 0) )
                       ( SETQ J (+ J 2) )
                  )
                  (COND ( (EQ FLAG 0)   NIL   )
                        (   T           T    ))
                  )
           )
    )
))
```

Example 6.

```
; --------------------------------------------------- ;
; Goldbach conjectured that every ;
; even number greater than or equal to 4 can be represented ;
; as the sum of two primes. This conjecture has ;
; not been proven or disproved to this day. ;
; Write a program to test this hypothesis for a given even ;
; number. The result of executing the program should be ;
; the output of the number itself if a pair of ;
; prime terms could not be found , and the output of a pair of corresponding ;
; primes if such a pair is found ;
; --------------------------------------------------- ;


(DEFUN SM (LAMBDA (N)
    (SETQ J 2)
    (LOOP
        ( (EQ J N) )
        ( (AND (SIMPLE J)
               (SIMPLE (- N J)))
               ( (PRIN1 J) (PRIN1 " ")
                 (PRINT (- N J)) )
        )
        (SETQ J (+ J 1))
    )
))
; --------------------- ;
(DEFUN SIMPLE (LAMBDA (N)
    (SETQ FLAG 1)
    (COND ( (AND (EQ (MOD N 2) 0) (NOT (EQ N 2))) NIL )
          (  T  ( (SETQ I 3)
                  (LOOP
                          ( (< (CAR (DIVIDE N 2)) I) )
                          ( (EQ (MOD N I) 0)
                               (SETQ FLAG 0) )
                          ( SETQ I (+ I 2) )
                     )
                     (COND ( (EQ FLAG 0) NIL )
                           (   T         T  ))
                     )
             )
       )
   ))
```

Example 7.

```
   ; ----------------------------------------------------- ;
   ; Determine whether a given integer p is prime, ;
   ; using Wilson's theorem: an integer p is ;
   ; prime if and only if (p-1)!+1 is divisible ;
   ; by p ;
   ; ----------------------------------------------------- ;
   (DEFUN PROST (LAMBDA (P)
      (COND  ( (ZEROP (MOD (FACT P) P))
                 (PRIN1 "Prime number") )
             ( T (PRIN1 "The number is not prime") )
      )
   ))
   ; --------------------- ;
   (DEFUN FACT (LAMBDA (NUM)
   ; Calculating (NUM-1)! + 1 ;
      (SETQ ANS 1)
      (SETQ NUM (- NUM 1))
      (LOOP
          ( (EQ NUM 0) ANS )
          (SETQ ANS (* NUM ANS))
          (SETQ NUM (- NUM 1))
      )
      (SETQ ANS (+ ANS 1))
   ))
```

In the next step we will begin to look at ***object-oriented programming in the*** **muLISP-87** system.

Step 108.
Object-oriented programming in muLISP-87. Message-oriented OOP

In this step, we will list ***the main principles underlying object-oriented programming***.

The prototype of object-oriented programming was a number of tools contained in the **SIMULA-67** language . But it took shape as an independent programming style in connection with the appearance of the **Smalltalk** language, originally intended for the implementation of machine graphics functions and developed by A. Kay. The first version of this language was created in 1972 and since then several of its versions have been built.

The roots of object-oriented programming go back to one of the branches of logic, in which the primary thing is not the relation (as for logical programming), but ***the object*** . Compared to predicate calculus, object-oriented logical systems have a more complex syntax and inference rules. Closely related to object-oriented programming is ***actor theory*** .

There are different opinions about whether the method of object formation is the main issue in object-oriented programming, which puts ***the object*** at the forefront. When considering object-oriented languages from the standpoint of constructing computational models, one can also consider that an essential point is ***the sending of messages*** as a mechanism for organizing the computational process [1, pp. 87-88]. However, we note that in the simplest case, ***sending a message*** is emulated ***by calling a method of*** one or another object.

The main features of object-oriented languages are [2]:

- presence of active ***objects*** (***actors***);
- formation of objects by ***inheritance of properties***;
- ***sending messages*** from object to object as a mechanism for organizing the computing process.

The essence of this programming style is expressed by the formula ***"object = data + procedures"***.

Thus, an object integrates some ***state*** (or data structure) and ***mechanisms for changing this state that*** are accessible only to it. In order to modify the state of some object, it is necessary to send it a corresponding ***message***. ***The action*** (or ***method***) performed (executed) by the addressee of the message concerns only him: other objects should not know how this object implements this or that function.

The combination of data and procedures in an object is called ***encapsulation***, and this property is inherent in object-oriented programming. Object-oriented programming languages (**TURBO Pascal, C++**), which are oriented toward objects rather than sending messages, also have ***polymorphism***, i.e. the ability to use methods with ***the same*** names to work with data ***of different*** types.

Many object-oriented languages have facilities for grouping objects into ***classes***. Objects belonging to one class are called its ***instances***. This allows procedures (in terms of object-oriented programming - ***methods***) applicable to all instances of a class to be stored in a single copy, only in the corresponding class. This approach has obvious advantages: the code is placed in only one place, which saves memory and ensures program modifiability.

A number of languages, including **Smalltalk-80, C++, TURBO Pascal,** and **Objective-C**, develop the concept of ***a class*** by allowing classes to be placed at nodes in a so-called ***inheritance tree . This is how a class hierarchy*** is constructed. An instance inherits the properties (methods) of its class, as well as all ***superclasses*** (superclasses) of that class. Methods defined in the root class of the tree are inherited by all classes and their instances.

Inheritance, along with encapsulation and polymorphism, is the most important property of object-oriented programming.

The tree-structured inheritance mechanism is somewhat limited, since in many cases the partitioning of properties according to a strict hierarchy cannot be achieved. Therefore, some languages allow classes that inherit properties from more than one immediate ***superclass***, causing the class hierarchy to become ***a directed acyclic graph***, rather than a tree. Such a mechanism is added as an extension to **Smalltalk. Similar mechanisms are also available in the LISP-** based **Flavors, CommonLoops**, and **CLOS** object systems.

Object-oriented languages are used mainly in the construction of models, including the creation of knowledge representation languages and the implementation of computer network protocols. The potential capabilities of object-oriented languages are much broader.

Notes .

1. *[3, p.125]. In object-oriented programming, along with* ***the object model of the Smalltalk*** *language, the* ***Hewitt model of actors is*** *distinguished. The models differ slightly from each other in the order of control. If in the object model an object usually* ***returns a response to the sender of the***

***message* (*Sender*)**, then in the model with executors **the object passes it to the next object**, which continues the calculations.*

2. *The described model of discrete objects interacting with each other is a good basis for **parallelism**. Depending on the specification of the object-oriented computing model, there are different approaches to parallelization. The most promising of them is the following.*

***An object (called an actor** in this case) sending a message does not wait for a response from its addressee, but continues to function. Therefore, messages must be able to queue up for the addressee. Many actors can be activated simultaneously, and all computations are expressed primarily in terms of their communications. The languages of this class are the **Astra** system, as well as **Act 1** and **Act 3** .*

---

(1) Futi K., Suzuki N. Programming languages and circuit design of VLSI. - M.: Mir, 1988. - 224 p.
(2) Bogumirsky B.S. User's Guide for Personal Computer: In 2 parts. Part 1. - St. Petersburg: OILCO Association, 1992. - 357 p.
(3) Hyvänen E., Seppänen J. The World of Lisp. In 2 volumes. Volume 2: Programming methods and systems. - M.: Mir, 1990. - 319 p.

From the next step we will start to consider ***the object-oriented system*** **FLAVOR**.

Step 109.
Object-oriented system FLAVOR. Flavors (classes)

In this step we will get acquainted with ***flavors***.

The idea of object-oriented programming arose in the processing of the properties of atoms long before the languages**Simula** and **Smalltalk** were born. However, it was not until the development of the corresponding programming languages that it developed into a "pure" programming paradigm (or ***metaphor***). Nevertheless, the capabilities inherent in the **LISP language** made it possible to assimilate and incorporate the results of other developments, often in a more sophisticated form than in the original language.

As an object system based on the **muLISP** dialect, we will consider the **Flavors** system, which is an object-oriented extension of the **Common LISP** dialect.

In object programming, a program is not assumed to consist of a set of procedures and data; its basic constituent units are ***objects***, which contain both procedures and data. The idea of combining data and procedures is the basic idea of the object model and the basis of object programming.

An object type is often called ***an Object Class* or** simply a ***class***.

A class is the set of all potential objects that can arise from its definition. Objects are divided into classes based on their inherent properties and the actions they possess. The properties and actions of objects of the same class are the same. The set of properties of objects of different classes is different, although some properties and actions may coincide. Interactions between objects of the same or different classes are defined based on the actions inherent to the class and the messages that correspond to them.

For objects, it is necessary to distinguish between ***definition*** and ***invocation***.

An object definition is an abstract description of an object type that represents all objects of that type.

By ***calling an object*** from a definition, ***instances of objects*** (actual objects) are created.

The Flavor *concept* corresponds to a class in the **Flavors** system. However, a Flavor is a more general concept than **a Smalltalk** class**,** since it can also represent some abstract properties that cannot be implemented by an object as such.

For example, the concept "round" cannot be realized by a specific independent object. Let us call such classes ***characteristic classes*** **(Mixing Flavor)** or ***property classes*** in contrast to ***object classes***, which we will call ***natural classes*** .

A characteristic class can be used in the definition of a class of objects so that it "flavors" objects of some ***base class of objects*** (**Base Flavor**). Objects can be provided with different "shades" and "flavors", quite similar to adding different seasonings to food.

If common properties can be found among classes of objects, they can be extracted and defined as a characteristic class. This can significantly reduce class definitions. Thus, the characteristic class is another mechanism for abstracting data and actions.

In the **Flavors system, *the definition of a flavor*** is as follows:

```
   (DEFFLAVOR FLAV VARS INH-FLAVS . OPTIONS)
```

The DEFFLAVOR form defines a flavor **FLAV** with the superclasses specified by the **INH-FLAVS** list, and places ***the instance variables*** specified in the **VARS** list in the flavor's property list.

Furthermore, the **DEFFLAVOR** form creates **:GET** and/or **:SET *methods*** if the key parameters

```
   :GETTABLE-INSTANCE-VARIABLES
              and (or)
   :SETTABLE-INSTANCE-VARIABLES
```

are included in the list of **OPTIONS *modes***.

For example:

```
   ; Definition of a flavor named BODY
   (DEFFLAVOR BODY                           ; Flavor name
      ; --------------------------------------------------
      (
        (X 0) (Y 0) (CENTER-X SUN-X)         ; Properties
        (CENTER-Y SUN-Y) ORBIT RADIUS        ; flaver
        (TRAIL NIL) (ANGLE 1) SPEED SIZE     ;
      )
      ; --------------------------------------------------
      ()                                     ; Superclasses
      ; --------------------------------------------------
      (:SETTABLE-INSTANCE-VARIABLES CENTER-X CENTER-Y)
                                             ; Modes
   )
```

The DEFFLAVOR form is implemented as a complex macro. A side effect of calling it is the automatic generation of the necessary ***access functions*** that can be used when reading and assigning values to variables.

The side effect that occurs in connection with the generation of an object can be controlled by means of the following key parameters (modes), which are optionally specified in the definition.

1. Mode

```
  :GETTABLE-INSTANCE-VARIABLES
```

causes the system to generate methods that read the values of instance variables.

2. Mode

```
  :SETTABLE-INSTANCE-VARIABLES
```

causes methods to be generated for assigning new values to instance variables.

Value assignment methods have the form

```
  :SET-x
```

where **x** is the name of the instance variable.

For example, the name of the method assigned to the variable ***NAME*** would be **:SET-NAME** .

Based on the definition of a Flaver, we can create new instances belonging to this Flaver. This is done by calling the macro

```
  (MAKE-INSTANCE FLAV . VARS)
```

which creates an instance of a flaver **FLAV** with a list of instance variables **VARS**.

For example:

```
  (MAKE-INSTANCE
     PLANET                ; Flavor name
     ; ------------------------------------------
     :ORBIT-RADIUS 114     ; Assigning values
     :SPEED 0.053          ; to properties (variables
     :SIZE 4               ; of the instance)
  )
```

Since we want to refer to the created object later, we assign its value to the **MARS** variable:

```
  (SETQ MARS (MAKE-INSTANCE
                PLANETS
                :ORBIT RADIUS 114
                :SPEED 0.053
                :SIZE 4)
  )
```

In the next step we will talk about ***properties and methods***.

Step 110.
Object-oriented system FLAVOR. Properties and methods

In this step we will look at ***the description of properties and methods***.

The data contained in an object is stored in variables called ***instance* variables** or ***attributes*. Instance variables are local variables of each object; they have a name and a** value. For objects of the same type, instance variables have the same names, but their values may, of course, differ.

The values of instance variables define ***the state*** of an object, which is unique and changes over time for each object. The state of an object can be examined and modified using ***the object's methods***.

The actions provided in an object are called ***methods*** **(Method)**. ***Methods*** are functional properties that define the use of an object as a function or procedure with a side effect. The set of methods is not fixed, so it can be expanded by defining new methods.

The definition of the flavor method is done using the **DEFMETHOD** form:

```
  (DEFMETHOD (FLAV . METH) ARGS . BODY)
```

The form associates the definition of the flavor **FLAV** with the method **METH**, which is called with the arguments **ARGS** specified by the lambda list. The method body **BODY** is an arbitrary **LISP** expression.

For example:

```
  ; Method definition :ROTATE
  (DEFMETHOD
     (BODY :ROTATE)                      ; Flavor and Method
     ; -----------------------------------------------------
     (RELATIVE-ANGLE MOVEMENT OLDX OLDY) ; Method Arguments
     ; -----------------------------------------------------
     (SETQ RELATIVE-ANGLE                ;
           SPEED MOVEMENT)               ;
     (LOOP                               ;
         ( (= RELATIVE-ANGLE 0) )        ; Body
                ...                        ...
         )                               ; methods
     )                                   ;
  )                                      ;
```

In object-oriented programming, computation is controlled primarily by methods calling each other, similar to procedure calls in traditional programming. However, the order in which methods are applied can be affected by their associated ***daemon methods***.

Daemon *Methods*, or ***daemons*** for short, are helper methods that are declared along with the main method definition. If the main method receives a message, the daemon is evaluated without mention.

There are two types of daemons [1, p.115-116]: ***pre-*** and ***post-daemons***. ***A*** **Before-Daemon** is a method that is always executed before the main method. ***An*** **After-Daemon** is executed after the main method.

With pre- and postdaemons, you can, for example, track when a certain message is sent or returned by an object, modify other objects based on the messages they receive or send, etc.

The type of daemon is expressed by the keywords **:BEFORE** and **:AFTER** (pre- and post-daemons).

For example:

```
  (DEFMETHOD
     (PLANET :AFTER :ROTATE)         ; Flavor and method
                                     ; with postdemon
     ; --------------------------------------------------
     (NEXT-SATELLITE)                ; Method arguments
     ; --------------------------------------------------
     (SETQ SATELLITE-LIST)           ;
     (LOOP                           ;
         ( (NULL S) )                ; Body
         (SETQ NEXT-SATELLITE        ;
               (POP S))              ; method
             ...                      ...
                                     ;
     )
  )
```

Method composition can be performed not only by time via pre- and postdemons, but also by other means. The method of method composition can be expressed, for example, by a function on the basis of which the calculation of methods associated with a known message or situation is determined. For example, the composition of methods using the **AND** function is calculated until the first method whose value is **NIL** is encountered.

---

(1) Hyvänen E., Seppänen J. The World of Lisp. In 2 volumes. Volume 2: Programming Methods and Systems. - Moscow: Mir, 1990. - 319 p.

In the next step we will look at ***messaging***.

Step 111.
Object-oriented system FLAVOR. Message exchange

In this step we will look at ***the implementation of messaging***.

Objects are individual, and each of them has its own current state and its own set of methods corresponding to the type. The method calls are generalized and their uniformity is ensured by a special mechanism called ***message exchange*** . Thus, an object method can be activated by sending ***a message* corresponding to this method (Message)**.

Sending a message to an object is accomplished by calling the universal function **SEND**, in which the object is passed the name of the method being called and the arguments it uses. The object receiving the message contains a mechanism that recognizes the message, selects the corresponding method, activates it, and passes it the parameters contained in the message. The method is evaluated in the same way as an ordinary function, and its value is returned as a response to the message.

So, the message is sent using the **SEND** form:

```
  (SEND OBJ METH . VARS)
```

OBJ is the object to which a **METH** message with **VARS** arguments is sent. The object receiving the message invokes the corresponding method and, generally speaking, returns the message to the object that sent the message.

For example:

```
  ; Let the planets spin!
  (SEND MERCURY : SPIN)
  (SEND VENUS : SPIN)
  (SEND EARTH : SPIN)
  (SEND MARS :SPIN)
  ; Assignment of satellites to Earth and Mars
  (SEND EARTH :SET-LIST-SATELLITES (LIST MOON))
  (SEND MARS :SET-LIST-SATELLITES (LIST PHOBOS DEIMOS))
```

If an object does not have a method corresponding to the received message, the message is lost. This is usually caused by an error or at least an exceptional situation that you need to be prepared to handle.

It is important to note that evaluating a method may, as a side effect, change the state of an object and cause new messages to be sent to other objects.

After a message is processed and the object is no longer active, ***it and its state are saved*** and the object will wait for new messages. Instance variables and their values do not disappear as in the case of regular functions, and their values are saved and can be used when the object receives a new message.

Objects, methods and message exchange allow the use of a new principle of program construction and calculations in it, more general and flexible than the traditional idea of a subroutine. The course of events is not controlled from the outside, but is carried out depending on the behavior of objects (their reaction to messages).

Objects are more independent units than traditional routines. They can contain information about their own structure and functioning, participate in the control of computation, and interact with other objects by sending and receiving messages.

Example 1 [1, pp.118-121]. The program is designed to use the **Flavor** object-oriented programming system, contained in the **FLAVORS.LSP** file (author: Peter Ohler), supplied with the **muLISP-87** system .

```
(DEFFLAVOR
   SHIP                           ; Flavor name
   (NAME X Y X-SPEED Y-SPEED)     ; Instance variables
   ()                             ; No superclasses
   :SETTABLE-INSTANCE-VARIABLES   ; Modes
   :GETTABLE-INSTANCE-VARIABLES
)
; --------
(DEFMETHOD
   (SHIP:SPEED)
   ()
   (+ (* X-SPEED X-SPEED)
      (* Y-SPEED Y-SPEED))
)
% ------------
(DEFUN MAIN ()
   (SETQ SHIP1 (MAKE-INSTANCE 'SHIP))
   ; We will give the object a name when creating it
   (SETQ SHIP78 (MAKE-INSTANCE 'SHIP :NAME 'Titanic))
   ; The object is given a new name
   (PRINT (SEND SHIP78 :SET-NAME 'BORE))
   ; The object is asked for its name
   (PRINT (SEND SHIP78 :NAME))
   ; The object is assigned X-SPEED
   (PRINT (SEND SHIP78 :SET-X-SPEED 3.0))
   ; The object is assigned Y-SPEED
   (PRINT (SEND SHIP78 :SET-Y-SPEED 4.0))
   ; The object is asked for SPEED
   (PRINT (SEND SHIP78 :SPEED))
)
; ----
(MAIN)
```

Example 2. A simple ***window system*** **written in an object-oriented style. The program is designed to use the Flavor** object-oriented programming system, contained in the **FLAVORS.LSP** file (author: Peter Ohler), supplied with the **muLISP-87** system.

```
(SETQ *ALL-WINDOWS*)
(DEFFLAVOR WINDOW
; Basic window flair
   (
      TOP                 ; Top of the window in rows
      LEFT                ; Left edge of the window in columns
      HEIGHT              ; Height in columns
      WIDTH               ; Width in rows
      (CURRENT-ROW 0)     ; Cursor position (row)
      (CURRENT-COL 0)     ; Cursor position (column)
      (EXPOSEDP NIL)      ; Is the window on (visible?)?
      (BORDER 1)          ; Width of the border around the window
      (BORDER-COLOR 7)    ; Border color
      (FOREGROUND 15)     ; Window foreground color
      (BACKGROUND 0)      ; Window background color
   )
   ()                     ; No superclasses
   :SETTABLE-INSTANCE-VARIABLES
   :GETTABLE-INSTANCE-VARIABLES
)
; ---------------------------------
(DEFMETHOD (WINDOW :AFTER :INIT) ()
; After creating a window, the method adds its name to the
; list of windows
   (PUSH SELF *ALL-WINDOWS*)
)
; ----------------------------
(DEFMETHOD (WINDOW:EXPOSE) ()
; The method allows you to highlight a window: it creates a window with
; the given borders, sets the cursor to the desired position
; if the window is already highlighted; otherwise
; it clears the window
   ((SEND SELF :EXPOSEDP)
     (MAKE-WINDOW (+ TOP BORDER)
                  (+ LEFT BORDER)
                  (- HEIGHT (* 2 BORDER))
                  (- WIDTH (* 2 BORDER))
     )
     (SET-CURSOR CURRENT-ROW CURRENT-COL)
   )
  (SEND SELF :REFRESH)
)
; -----------------------------
(DEFMETHOD (WINDOW :REFRESH) ()
; The method "clears" the window with the required color and sets
; the variables displayed on the screen to TRUE
   (BACKGROUND-COLOR BORDER-COLOR)
   (FOREGROUND-COLOR 0)
   (MAKE-WINDOW TOP LEFT HEIGHT WIDTH)
   (CLEAR-SCREEN)
   (BACKGROUND-COLOR BACKGROUND)
   (FOREGROUND-COLOR FOREGROUND)
   (MAKE-WINDOW (+ TOP BORDER)
                (+ LEFT BORDER)
                (- HEIGHT (* 2 BORDER))
                (- WIDTH (* 2 BORDER))
   )
   (CLEAR-SCREEN)
   (SETQ CURRENT-ROW 0 CURRENT-COL 0)
   (SETQ EXPOSEDP 'T)
)
; --------------------------------------------
(DEFMETHOD (WINDOW :AFTER :REFRESH) (WINDOWS)
; After the window is cleared, the method checks all remaining
; windows and if they are overlapped by the new window, then
; highlights them again
   (SETQ WINDOWS (REMOVE SELF *ALL-WINDOWS*))
```

```
      (LOOP
         ( (NULL WINDOWS) )
         ( ( (NO-OVERLAP TOP LEFT HEIGHT WIDTH (CAR WINDOWS))
                (POP WINDOWS)
           )
           (SEND (CAR WINDOWS) :SET-EXPOSEDP NIL)
           (POP WINDOWS)
         )
      )
   )
   ; ---------------------------------------------------
   (DEFMETHOD (WINDOW :BEFORE :PRINT) (STR ROW COLUMN)
   ; Before you can display anything in a window, it must
   ; appear
      (SEND SELF :EXPOSE)
   )
   ; ---------------------------------------------------
   (DEFMETHOD (WINDOW :BEFORE :PRINC) (STR ROW COLUMN)
   ; Before you can display anything in a window, it must
   ; appear
      (SEND SELF :EXPOSE)
   )
   ; -------------------------------------------
   (DEFMETHOD (WINDOW :PRINT) (STR ROW COLUMN)
   ; Prints STR in the given ROW and COLUMN of the window
      (IF (AND (INTEGERP ROW) (INTEGERP COLUMN))
         (SET-CURSOR ROW COLUMN))
      (PRINT STR)
      (SETQ CURRENT-ROW (ROW) CURRENT-COL (COLUMN))
   )
   ; -------------------------------------------
   (DEFMETHOD (WINDOW :PRINC) (STR ROW COLUMN)
   ; Prints STR in the given ROW and COLUMN of the window
      (IF (AND (INTEGERP ROW) (INTEGERP COLUMN))
         (SET-CURSOR ROW COLUMN))
      (PRINC STR)
      (SETQ CURRENT-ROW (ROW) CURRENT-COL (COLUMN))
   )
   ; --------------------------
   (DEFWHOPPER (WINDOW :PRINT)
                        (STR ROW COLUMN W1 W2 W3 W4 RC)
   ; We want to return to the current window after working in
   ; another window, so we find the dimensions of the current
   ; window and then reset them after
   ; executing the :PRINT method
      (SETQ R (MAKE-WINDOW) W1 (POP R) W2 (POP R)
            W3 (POP R) W4 (POP R))
      (SETQ R (ROW) C (COLUMN))
      (IF (NO-OVERLAP W1 W2 W3 W4 SELF)
         (CONTINUE-WHOPPER STR ROW COLUMN))
      (MAKE-WINDOW W1 W2 W3 W4)
      (SET-CURSOR RC)
   )
   ; --------------------------
   (DEFWHOPPER (WINDOW :PRINC)
                         (STR ROW COLUMN W1 W2 W3 W4 RC)
   ; We want to return to the current window after working in
   ; another window, so we find the dimensions of the current
   ; window and then reset them after
   ; executing the :PRINC method
      (SETQ R (MAKE-WINDOW) W1 (POP R) W2 (POP R)
            W3 (POP R) W4 (POP R))
      (SETQ R (ROW) C (COLUMN))
      (IF (NO-OVERLAP W1 W2 W3 W4 SELF)
         (CONTINUE-WHOPPER STR ROW COLUMN))
      (MAKE-WINDOW W1 W2 W3 W4)
      (SET-CURSOR RC)
   )
   ; --------------------------------------------------- -
   (DEFUN NO-OVERLAP (TOP1 LEFT1 HEIGHT1 WIDTH1 WINDOW2)
   ; Does the window overlap the rectangle with sides
   ; formed by TOP1 LEFT1 HEIGHT1 WIDTH1
      ( (LAMBDA (TOP2 LEFT2 HEIGHT2 WIDTH2
                 BOTTOM1 BOTTOM2 RIGHT1 RIGHT2)
           (SETQ BOTTOM1 (+ TOP1 HEIGHT1)
                 BOTTOM2 (+ TOP2 HEIGHT2)
                 RIGHT1 (+ LEFT1 WIDTH1)
                 RIGHT2 (+ LEFT2 WIDTH2))
           ( (OR (<= TOP1 TOP2 BOTTOM1)
                 (<= TOP1 BOTTOM2 BOTTOM1)
                 (<= TOP2 BOTTOM1 BOTTOM2))
             ( (OR (<= LEFT1 LEFT2 RIGHT1)
                   (<= LEFT1 RIGHT2 RIGHT1)
                   (<= LEFT2 RIGHT1 RIGHT2))
               NIL))
           'T
        )
        (SEND WINDOW2 :TOP)
        (SEND WINDOW2 :LEFT)
        (SEND WINDOW2 :HEIGHT)
        (SEND WINDOW2 :WIDTH)
      )
   )
   ;-----------------------------------------
   ; Making a copy of the WINDOW flaver
   (SETQ SMALL (MAKE-INSTANCE 'WINDOW
                              :TOP 2
                              :LEFT 10
                              :HEIGHT 8
                              :WIDTH 60))
   ; ----------------------------------------
   ; Making a copy of the WINDOW flaver
   (SETQ TINY (MAKE-INSTANCE 'WINDOW
                              :TOP 3
                              :LEFT 30
                              :HEIGHT 8
                              :WIDTH 20
                              :BORDER 2))
   ; ----------------------------------------
   ; Making a copy of the WINDOW flaver
   (SETQ MAIN (MAKE-INSTANCE 'WINDOW
                             :TOP 12
                             :LEFT 0
                             :HEIGHT 12
                             :WIDTH 80))
   ; -------------------------------------------
   (SEND MAIN :EXPOSE)      ; Enable MAIN window
   (WRITE-LINE
     "Message sent (SEND SMALL :PRINC 'HELLO 2 2)")
   (SEND SMALL:PRINC 'HELLO 2 2)
   (WRITE-LINE
     "Message sent (SEND TINY :PRINC 'HELLO 2 2) ")
   (SEND TINY :PRINC 'HELLO 2 2)
   (SEND MAIN :REFRESH)      ; Cleared the MAIN window
   (SEND SMALL :REFRESH)      ; Cleared the SMALL window
   (SEND TINY :REFRESH)      ; Cleared the TINY window
```

*__Note:__ Sending a message corresponds to calling the **FUNCALL**. If we imagine that the definition of the **METHOD** function is stored in the property list of the **OBJECT** object under the name **METHOD**, then it could be called by the function:*

```
   (FUNCALL (GET OBJECT METHOD) ARGUMENTS)
```

*This makes the connection between object-oriented thinking and the property list explicit. The property list of a symbol is in a sense also an object. Its methods are, for example, the functions **GET, PUT, REMPROP**.*

---

(1) Hyvänen E., Seppänen J. The World of Lisp. In 2 volumes. Volume 2: Programming Methods and Systems. - Moscow: Mir, 1990. - 319 p.

In the next step we will talk about ***non-following***.

Step 112.
Object-oriented system FLAVOR. Inheritance

In this step we will look at ***the implementation of the inheritance mechanism***.

Consider two classes **A** and **B**. Denote by **P(A)** the set of instance variables and methods of class **A**, and **P(B)** the set of instance variables and methods of class **B** . The sets **P(A)** and **P(B)** may be in the following relationships:

1. **P(A)** and **P(B)** ***are disjoint***, i.e. the class objects do not share instance variables and methods.
2. **P(A)** and **P(B)** ***intersect***, so they have both common and distinct instance variables and methods.
3. **P(A)** ***contains*** the set **P(B)** or vice versa.

If **P(A)** contains the set **P(B)**, we will say that class **A** is ***a subclass*** **(Subclass)** of class **B** and, accordingly, class **B** is ***a superclass*** **(Superclass)** of class **A.** If instance variables and methods coincide, then we are dealing with the same class.

Thus, a subclass always has more instance variables and methods than the superclass that contains it. Its objects are, for example, more ***specific*** with respect to some instance variables than the objects of the superclass, which in turn are more ***general***.

It is now clear that a special mechanism is needed to form the hierarchical structure of classes. This purpose is served by ***the Inheritance Mechanism*****,** which is contained in one form or another in all object-oriented systems.

Through the mechanism of inheritance, a newly defined class can ***automatically inherit*** the instance variables and methods of the class defined as its superclass. With inheritance, it is achieved that more general properties associated with a higher-level type do not need to be specifically defined in relation to lower-level types. It is sufficient to specify from which superclass they are inherited.

For example:

```
   ; Definition of a flavor named BODY
   (DEFFLAVOR BODY                          ; Name of the flavor
      (
        (X 0) (Y 0) (CENTER-X SUN-X)        ; Properties
        (CENTER-Y SUN-Y) ORBIT RADIUS       ; flaver
        (TRAIL NIL) (ANGLE 1) SPEED SIZE    ;
      )
      ()                                       ; Superclasses
      (:SETTABLE-INSTANCE-VARIABLES CENTER-X CENTER-Y)
                                               ; Modes
   )
```

Planets and satellites are defined as a subclass of the ***BODY flower***

```
   (DEFFLAVOR PLANET                  ; Flavor name
      ( (SATELLITE-LIST NIL) )       ; Flavor properties
      (BODY)                          ; Superclasses
      :SETTABLE-INSTANCE-VARIABLES    ; Modes
      :GETTABLE-INSTANCE-VARIABLES
   )
   ; -----------------
   (DEFFLAVOR SATELLITE
      ()                              ; Flavor Properties
      (BODY)                          ; Superclasses
   )
```

This way, you can simplify object definitions and save effort on their maintenance. Changes can be localized to only one variable place in the program.

Different systems have different conventions about what ***the topology of inheritance structures*** can be . For example, in **Smalltalk, a class can have only one superclass, but several subclasses. In this case, only *tree*** **-** like inheritance structures are possible.

The Flavor system allows cyclic definitions, but excludes recursive inheritance. The structure of definitions can be ***a directed graph***, but the system recognizes cyclic definitions and breaks inheritance in this case. Each property is considered only once.

Note that even such diverse inheritance methods may be limited. For example, recursive types are used quite often in mathematics and natural language. However, for most practical applications, acyclic structures are sufficient. In this case, a class inherits the properties of not only the classes immediately above it, but also the properties of their superclasses, i.e. the properties of all classes located above it.

If a class inherits properties from many superclasses, then the question of the order of inheritance arises when properties of the same name are inherited from superclasses located in different branches, or if the same class occurs in the structure several times. In what order are the classes located and in what order are the properties taken into account?

In principle, possible search sequences are, for example, **Depth First *and*** Level ***First.*** In each branch, possible traversal orders are ***direct*** and ***reverse***. ***Direct order*** is the order in which a tree (or graph) node is traversed before its subtrees are traversed in the same order. ***Reverse order*** can be defined similarly.

In the **Flavor system,** ***depth-first search*** and ***direct order traversal*** are used to traverse the flavor hierarchy. Properties are inherited in the order in which superclasses are specified in the class description. A class with the same name is not traversed again. Let us assume that the class hierarchy looks like this [1, pp.114-115]:

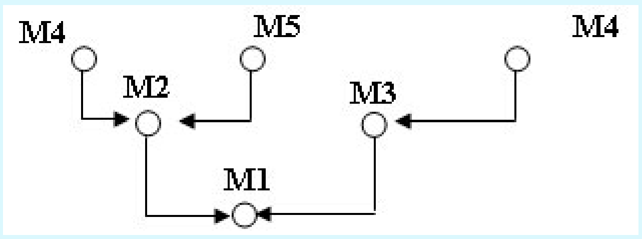


Fig.1. Class hierarchy

In this case, the order of passage would be as follows:

```
M1 -->M2 -->M4 -->M5 -->M3
```

Please note that class **M4** can only be completed once.

Since different classes may have methods with the same name but different definitions, to avoid conflicts it is necessary to agree on which of them will remain in force. Here it is reasonable to limit yourself to two possibilities: either the first method or the last one remains in force.

In the **Flavor system, *the first property encountered*** is the one that is in effect. If, for example, class **M5** had a method with the same name as class **M4**, then method **M5** would not be considered. The system does, however, allow you to change the default order. If, for example, it is determined that the last method is in effect, then it is said to ***override*** previous occurrences.

Subclass instance variables are defined as a set of its instance variables and instance variables of superclasses. If a superclass of a class has an instance variable with the same name, then it is ***shared for the class*** **(Shared Instance Variable)**. Shared variables can be used for communication between classes. The same subclass variable can, for example, be assigned and read by methods defined in different superclasses.

In addition to user-defined objects and classes, an object system typically contains common system classes whose properties are accessible to everyone. We call such classes ***base classes***.

For example, the Flavor system has ***a base Vanilla Flavor.*** Its properties and methods are inherited automatically by all objects.

The base class can contain various system properties and actions, such as:

- self-reference (**SELF**),
- reference to the class name and definition,
- methods that call an object and describe its default properties (**Default Method**).

Example. The program "Solar System" [1, p.4.5] for **muLISP-87**. It is designed to use the **Flavor** object-oriented programming system, contained in the **FLAVORS.LSP** file (author: Peter Ohler), supplied with the **muLISP-87** system .

```
; An astronomical model that represents and makes more
; visual the structure and functioning of a "piece"
of the solar ; system. The model describes the Sun and the planets Mercury,
; Venus, Earth, Mars and their satellites.
(SETQ SUN-X 150 SUN-Y 100)
(SETQ SUN-DIAMETER 10 EARTH-ORBIT 149)
(SETQ RED (+ 2 128))
                      ;Red color using XOR
(SETQ LIGHT (+ 1 128))
                      ;Cyan color using XOR
; -------------------------------------------------------
(DEFFLAVOR BODY
; Definition of a base class named BODY
   (
      (X 0)
      (And 0)
      (CENTER-X SUN-X)
      (CENTER-Y SUN-Y)
      RADIUS-ORBITS
      (AFTER NIL)
      (ANGLE 1)
      SPEED
      SIZE
   )
   ()
   (:SETTABLE-INSTANCE-VARIABLES CENTER-X CENTER-Y)
)
; ------------------------------------------
(DEFFLAVOR PLANET ((SATELLITE LIST NIL))
; Planets are defined as a subclass of the BODY class
   (BODY)
   :SETTABLE-INSTANCE-VARIABLES
   :GETTABLE-INSTANCE-VARIABLES
)
; --------------------
(DEFFLAVOR SATELLITE ()
; Satellites are defined as a subclass of the BODY class
   (BODY)
)
; -------------------------
(DEFMETHOD (BODY :ROTATE)
; Definition of method :ROTATE of class BODY
   (RELATIVE-ANGLE MOVEMENT OldX OldY)
   (SETQ RELATES-ANGLE SPEED DISPLACEMENT)
   (LOOP
      ( (= RELATIVE-ANGLE 0) )
      (IF (> RELATIVE-ANGLE 10)
         (SETQ RELATES-ANGLE (- RELATES-ANGLE 10)
               MOVEMENT 10)
         (SETQ MOVEMENT REF-ANGLE REF-ANGLE 0)
      )
      (SETQ ANGLE (+ ANGLE MOVEMENT))
      (SETQ OLDX X OLDY Y)
      (SETQ X (TRUNCATE (+ CENTER-X (* (SIN-DEG ANGLE)
                                       ORBITS RADIUS))))
      (SETQ Y (TRUNCATE (+ CENTR-Y (* (COS-DEG UGOL)
                                       ORBITS RADIUS))))
      (IF  (OR (/= X OLDX) (/= Y OLDY))
         (PROG1
            (PLOT-CIRCLE OLDX OLDY SIZE RED)
                                         ; Erase circle
            (PLOT-CIRCLE X Y SIZE RED)
                                         ; Red circle
            (IF TRACE (PLOT-DOT X Y LIGHT) NIL)
         )
      )
   )
)
; -------------------------------------------------------
(DEFMETHOD (PLANET :AFTER :ROTATE) (NEXT-SATELLITE)
   (SETQ S LIST-SATELLITES)
   (LOOP
      ( (NULL S) )
      (SETQ ANOTHER-SATELLITE (POP S))
      (SEND NEXT-SATELLITE :SET-CENTER-XX)
      (SEND NEXT-SATELLITE :SET-CENTER-YY)
      (SEND ANOTHER-SATELLITE: ROTATE)
   )
)
; ---------------------------------------------
; Trigonometric functions of the muLISP-87 system
; ---------------------------------------------
(DEFUN SIN-YOU (ANGLE)
; Returns the sine of the argument,
; expressed in degrees
   ( (MINUSP ANGLE)
           (SETQ ANGLE (DIVIDE (REM (- ANGLE) 360) 45))
      (- (SIN-COS-DEG (CAR ANGLE) (CDR ANGLE))) )
      (SETQ ANGLE (DIVIDE (REM ANGLE 360) 45))
      (SIN-COS-DEG (CAR ANGLE) (CDR ANGLE))
   )
   ; --------------------
   (DEFUN COS-DEG (ANGLE)
   ; Returns the cosine of the argument,
   ; expressed in degrees
      (SETQ ANGLE (DIVIDE (REM (ABS ANGLE) 360) 45))
      (SIN-COS-DEG (+ 2 (CAR ANGLE)) (CDR ANGLE))
   )
   ; -------------------------------
   (DEFUN SIN-COS-DEG (N45DEG RESID)
      ((> N45DEG 3)
      (- (SIN-COS-DEG (- N45DEG 4) RESID))
      ( (ZEROP N45DEG) (REDUCED-SIN RESID) )
      ( (EQ N45DEG 1) ( (ZEROP RESID) 0.70710678 )
      (REDUCED-COS (- 45 RESID)))
      ( (EQ N45DEG 2) (REDUCED-COS RESID) )
      ( (ZEROP RESID) 0.70710678 )
      (REDUCED-SIN (- 45 RESID))
   )
   ; ----------------------
   (DEFUN REDUCED-SIN (DEG)
      (/ (* YOU (+ 1324959969 (* (SETQ YOU (* YOU))
                                 (+ -67245 YOU))))
     75914915920)
   )
   ; ----------------------
   (DEFUN REDUCED-COS (DEG)
      (SETQ DEG (* DEG DEG))
      (/ (+ 266153374 (* YOU (+ -40518 YOU)))
         266153374) )
   ; -------------------------
   (DEFUN SOLAR ()
   ; Main function
      (SETQ MARS
            (MAKE-INSTANCE PLANET
                           :ORBIT RADIUS 114
                           :SPEED 0.053
                           :SIZE 4)
      )
      (SETQ PHOBOS
            (MAKE-INSTANCE SATELLITE
                           :CENTER-X 48
                           :CENTR-Y 48
                           :ORBIT RADIUS 7
                           :SPEED 114.4
                           :SIZE 3)
      )
      (SETQ MERCURY
            (MAKE-INSTANCE PLANET
                           :ORBIT RADIUS 29
                           :SPEED 0.416
                           :SIZE 3)
      )
      (SETQ VENUS
            (MAKE-INSTANCE PLANET
                           :ORBIT RADIUS 54
                           :SPEED 0.416
                           :SIZE 5)
      )
      (SETQ LAND
            (MAKE-INSTANCE PLANET
                            :ORBIT RADIUS 84
                            :SPEED 0.1
                            :SIZE 6)
    )
    (SETQ MOON
          (MAKE-INSTANCE SATELLITE
                            :CENTER-X 48
                            :CENTR-Y 48
                            :ORBIT RADIUS 15
                            :AFTER T
                            :SPEED 1.3
                            :SIZE 3)
    )
    (SETQ DEIMOS
          (MAKE-INSTANCE SATELLITE
                            :CENTER-X 48        ;(SEND MARS :X)
                            :CENTER-Y 48        ;(SEND MARS :Y)
                            :ORBIT RADIUS 12
                            :SPEED 30.4
                            :SIZE 3)
    )
    ; "Assignment" of satellites to Earth and Mars
    (SEND EARTH :SET-LIST-SATELLITES (LIST MOON))
    (SEND MARS :SET-LIST-SATELLITES (LIST PHOBOS DEIMOS))
    ; Launch of the Solar System
    (VIDEO-MODE 4)
    (PLOT-CIRCLE SUN-X SUN-Y
                 (- SUN-DIAMETER 3) RED)
    (LOOP
       (PLOT-CIRCLE SUN-X SUN-Y
                    (- SUN-DIAMETER 3) RED)
       (PLOT-CIRCLE SUN-X SUN-Y
                    (- SUN-DIAMETER 1) RED)
       (PLOT-CIRCLE SUN-X SUN-Y
                    DIAMETER-SUN RED)
       ; Let the planets spin!
       (SEND MERCURY : SPIN)
       (SEND VENUS : SPIN)
       (SEND EARTH : SPIN)
       (SEND MARS :SPIN)
    )
 )
 (SOLAR)
```

***Note* :** *Here is the algorithm for executing this library.*

1. *Load the **muLISP-87** programming environment (file **l.com**) and the **Flavor** system (file **flavors.lsp**) by running the following command in the command line:*

```
l.com flavors.lsp
```

2. *When asked whether to start the demonstration, answer **No** (press the **N** key).*
3. *Load the contents of the file **l112_1.lsp**, where the above program is located:*

```
(LOAD L112_1.LSP)
```

Once the download is complete, the program will start running automatically.

4. *To exit the program, press the **ESC** key and then follow the system prompts.*

In addition, abstract sets of properties can be distinguished into ***characteristic classes*** and used in definitions. A characteristic class can contain properties that are opposite or similar to the properties of the base classes in their natural classification.

For example, the classes "sun" and "apple" might inherit the property "round" from some class of geometric properties.

The most important advantage of object programming is perhaps that it offers an abstraction mechanism for breaking down a problem and its solution into parts. On the other hand, it assumes that the objects of the problem domain, as well as their properties and dependencies, must be carefully analyzed.

To distinguish classes, one can use the natural division of objects into different classes. For example, living beings are divided into mammals, birds, reptiles, fish, insects, etc. They are then subdivided into subclasses according to additional properties of each type.

---

(1) Hyvänen E., Seppänen J. The World of Lisp. In 2 volumes. Volume 2: Programming Methods and Systems. - Moscow: Mir, 1990. - 319 p.

From the next step we will provide ***tasks for independent solution***.

Step 113.
Practical lesson #3. Fundamental data types. Lists

In this step we will introduce ***list processing tasks***.

Example 1. A function that determines whether the first and last elements of a list **L** consisting of atoms are equal.

Solution:

```
(DEFUN RAWN (LAMBDA (L)
   (COND ( (NULL L) NIL )
         ( (EQ (LENGTH L) 1) NIL )
         ( T  (COND ( (EQ (CAR L) (LAST (CDR L))) T )
                    ( T  NIL )
              )
         )
   )
))
;-------------------- ;
(DEFUN LAST (LAMBDA (L))
   (COND ( (NULL L) NIL )
         ( (NULL (CDR L)) (CAR L) )
         (  T  (LAST (CDR L)) )
   )
))
```

Test examples:

```
    $ (RAWN '(1 2 3))
    NIL
    $ (RAWN '(1 2 3 4 5 1))
    T
    $ (SEE '((1) 2 3 4 5 (1)))
    T
```

Note that the given solution is ***not optimal****. Try to improve it!*

Example 2.

```
 (DEFUN NTH (LAMBDA (N LST)
  ; The function returns the N-th element of the list LST ;
    (COND ( (EQ N 1) (CAR LST) )
          (    T      (NTH (- N 1) (CDR LST) ) )
    )
 ))
 ; ------------------------- ;
 (DEFUN DELETE (LAMBDA (N LST)
 ; The function removes the N-th element from the list LST ;
    (COND ( (EQ N 1) (CDR LST) )
          (  T   (CONS (CAR LST)
                 (DELETE (- N 1) (CDR LST))) )
    )
 ))
 ; --------------------------- ;
 (DEFUN INSERT (LAMBDA (X N LST)
 ;The function inserts element X at the N-th position ;
 ;into the list LST ;
    (COND ( (NULL LST) (CONS X LST) )
          ( (EQ N 1) (CONS X LST) )
          (  T   (CONS (CAR LST)
                       (INSERT X (- N 1)
                               (CDR LST))) )
    )
 ))
```

Example 3.

```
 (DEFUN SORTING (LAMBDA (L)
  ; Sorting unordered list L ;
    (COND ( (NULL L) NIL )
          (   T    (INSERT (CAR L) (SORTING (CDR L))) )
    )
 ))
 ; ----------------------- ;
 (DEFUN INSERT (LAMBDA (AL))
 ; The INSERT function adds an element A to an ordered ;
 ; list L so that the order is preserved.
    (COND ( (NULL L) (LIST A) )
          ( (< A (CAR L)) (CONS A L) )
          (   T    (CONS (CAR L) (INSERT A (CDR L))) )
    )
 ))
 ; -------------------------- ;
 (DEFUN RASSTAV (LAMBDA (NEW L)
 ; The RASSTAV function allows elements of the NEW list
 ; to be inserted into an ordered list L ;
```

```
    (COND ( (NULL NEW) L )
          (  T  (INSERT (CAR NEW) (RASSTAV (CDR NEW) L)) )
    )
))
```

Test examples:

```
$ (SORTING '(5 4 3 2 1 6))
(1 2 3 4 5 6)
$ (RASSTAV '(4 3 1) '(1 2 5 6))
(1 1 2 3 4 5 6)
```

Example 4. A function that returns a list of all atoms that appear in a given list **LST**, along with their frequency of occurrence. Since the result is a set of pairs, the order in which the pairs are arranged is irrelevant. For example:

```
(NEITHER THIS NOR THE OTHER) --> ((NEITHER 2) (THEN 1) (OTHER 1)).
```

Solution:

```
(DEFUN KOLAM (LAMBDA (LST)
   (TH (LIST-SET LST) LST)
))
; ------------------------- ;
(DEFUN LIST-SET (LAMBDA (LST)
; The LIST-SET function converts an list LST into a set ;
   (COND ( (NULL LST) NIL )
         ( (MEMBER (CAR LST) (CDR LST))
             (LIST-SET (CDR LST)) )
         ( T (CONS (CAR LST) (LIST-SET (CDR LST))) )
   )
))
; ------------------------ ;
(DEFUN TH (LAMBDA (LST LST0)
; Construct a new list containing elements of the form ;
; (Element Number_of_repetitions_of_this_element_in_list
) ;
   (COND ( (NULL LST) NIL)
         (  T  (CONS (LIST (CAR LST)
                           (KOL (CAR LST) LST0))
                     (TH (CDR LST) LST0)) ))
))
; ---------------------- ;
(DEFUN COL (LAMBDA (M LST)
; Count the number of repetitions of element M in the list LST ;
   (COND ( (NULL LST) 0 )
         ( (EQ (CAR LST) M)
              (+ (KOL M (CDR LST)) 1) )
         (  T  (KOL M (CDR LST)))
   )
))
```

Test examples:

```
$ (POOL '(1 2 1 2 3 3 3 4))
((1 2) (2 2) (3 3) (4 1))
$ (POOL '(1 5))
((1 1) (5 1))
```

Example 5. A function that rearranges the elements of a list so that identical elements become neighbors. For example:

```
   (A B C A B B) --> (A A B B B C)
```

Solution:

```
   (DEFUN SOSEDI (LAMBDA (LST)
      (COND ( (NULL LST) NIL )
            (  T  (COND ( (MEMBER (CAR LST) (CDR LST))
                           (CONS (CAR LST)
                                 (NEIGHBORS
                                   (CONS (CAR LST)
                                         (DELETEFIRST
                                           (CAR LST)
                                           (CDR LST))))) )
                        (  T  (CONS (CAR LST)
                              (NEIGHBORS (CDR LST))) )
                  )
            )
      )
   ))
   ; ------------------------------ ;
   (DEFUN DELETEFIRST (LAMBDA (A LST)
   ; Remove the first occurrence of element A from the list LST ;
      (COND ( (NULL LST) NIL )
            ( (EQUAL (CAR LST) A) (CDR LST) )
            (   T  (CONS (CAR LST)
                         (DELETEFIRST A (CDR LST))) ))
   ))
```

Test examples:

```
   $ (NEIGHBORS '(1 2 3 11 1 3))
   (1 1 2 3 3 11)
   $ (NEIGHBORS '(0 1))
   (0 1)
   $ (NEIGHBORS '())
   NIL
```

Example 6. A predicate that checks whether the sequence of elements of a given numeric list is sorted in ascending order.

Solution:

```
   (DEFUN UPV (LAMBDA (LST)
      (COND ( (EQUAL LST (SORTING LST)) PRIN1 "Yes" )
            ( T PRIN1 "No")
      )))
   ; ------------------------ ;
   (DEFUN SORTING (LAMBDA (LST)
   ; Sorting an unordered list ;
      (COND ( (NULL LST) NIL )
            (  T  (INSERT (CAR LST) (SORTING (CDR LST)))  )
      )
   ))
   ; ------------------------- ;
   (DEFUN INSERT (LAMBDA (A LST)
   ; Adds element A to the ordered list LST so that ;
   ; the ordering is preserved.
```

```
    (COND ( (NULL LST) (LIST A) )
          ( (< A (CAR LST)) (CONS A LST) )
          (   T    (CONS (CAR LST) (INSERT A (CDR LST))) )
    )
))
```

Test examples:

```
$ (UPV '(1 2 3 4))
Yes
$ (UPV '(1 2 3 1))
No
$ (UPV '())
Yes
```

Example 7. ***A filter*** is a function that leaves or removes elements that satisfy a given condition.

The filters are defined below:

```
1) (DELETE-IF-PREDICAT LST)
2) (DELETE-IF-NON-PREDICAT LST),
```

removing from the list elements that have or do not have the property whose presence is tested by the predicate **PREDICAT**.

Solution:

```
 (DEFUN DELETE-IF-PREDICATE (LAMBDA (LST)
    (COND ( (NULL LST) NIL )
          ( (EQ (PREDICAT (CAR LST)) F )
                 (CONS (CAR LST)
                       (DELETE-IF-PREDICAT (CDR LST))) )
          (  T  (DELETE-IF-PREDICAT (CDR LST)) )
    )
))
; --------------------------------------- ;
 (DEFUN DELETE-IF-NOT-PRECIDES (LAMBDA (LST)
    (COND ( (NULL LST) NIL )
          ( (EQ (PREDICAT (CAR LST)) T )
                 (CONS (CAR LST)
                       (DELETE-IF-NON-PREDICAT (CDR LST))) )
          (  T  (DELETE-IF-NON-PREDICAT (CDR LST))  )
    )
))
; ----------------------- ;
 ( DEFUN PREACH (LAMBDA ( X )
; The predicate checks the equality of the element 0 ;
    (COND ( (EQ X 0) T )
          ( T  F )
    )
))
```

Test examples:

```
$ (DELETE-IF-PREDICATE '(1 0 4 0 5))
 (1 4 5)
$ (DELETE-IF-PREDICT '())
NIL
$ (DELETE-IF-PREDICT '(0 0))
NIL
$ (DELETE-IF-NON-PREDICAT '(1 0 3 0 5))
```

```
(0 0)
$ (DELETE-IF-NON-PREDICAT '(0 0 0))
(0 0 0)
$ (DELETE-IF-NON-PREDICAT '(1 5))
NIL
$ (DELETE-IF-NON-PREDICAT '())
NIL
```

Example 8. The function searches a list of atoms for an atom that occurs immediately before a given atom.

Solution:

```
(DEFUN PATOM (LAMBDA (X LST)
   (NTH (- (NATOM X LST) 1) LST)
))
; ---------------------- ;
(DEFUN NTH (LAMBDA (N LST)
; The function returns the N-th element of the list LST ;
    (COND ( (EQ N 1) (CAR LST) )
          (    T     (NTH (- N 1) (CDR LST)) )
    )
))
; ------------------------ ;
(DEFUN NATOM (LAMBDA (X LST)
; The function returns the position of the specified element ;
   (COND ( (NULL LST) 0)
         ( (EQ X (CAR LST)) 1)
         ( (MEMBER X LST) (+ 1 (NATOM X (CDR LST))) )
   )
))
```

Test examples:

```
$ (PATOM 5 '(9 7 2 7 5 3))
7
$ (PATOM 2 '(9 7 2 7 5 3))
7
$ (PATOM 3 '(9 7 2 7 5 3))
5
```

Example 9.

```
(DEFUN SWAP (LAMBDA (N M LST)
 ; Swap the values of the N-th and M-th elements of the list LST ;
   (INSERT (NTH N LST) M
           (DELETE M (INSERT (NTH M LST) N
                             (DELETE N LST))))
))
; ---------------------- ;
(DEFUN NTH (LAMBDA (N LST)
; The function returns the N-th element of the list LST ;
   (COND ( (EQ N 1) (CAR LST) )
         (    T     (NTH (- N 1) (CDR LST)) )
   )
))
; ---------------------------- ;
(DEFUN INSERT (LAMBDA (X N LST)
;The function inserts element X at the N-th position ;
;into the list LST ;
   (COND ( (NULL LST) (CONS X LST) )
          ( (EQ N 1) (CONS X LST) )
          (  T  (CONS (CAR LST)
                      (INSERT X (- N 1)
                              (CDR LST))) )
      )
  ))
  ; ------------------------- ;
  (DEFUN DELETE (LAMBDA (N LST)
  ; The function removes the N-th element from the list LST ;
     (COND ( (EQ N 1) (CDR LST) )
          (  T  (CONS (CAR LST)
                (DELETE (- N 1) (CDR LST))) )
     )
  ))
```

Tasks for independent solution

1. Define the **DISTL** function (distribution to the left), the operation of which we will consider using an example:

```
(A (B1 B2 ... BN)) --> ((A B1) (A B2) ... (A BN))
```

2. Define the DISTR function (distribution right), the action of which we will consider using an example:

```
((A1 A2 ... AN) B) -- > ((A1 B) (A2 B) ... (AN B)).
```

3. Define a function **A** such that **(A N)** returns the list **(THE ANSWER IS N)**. So the value of **(A 12)** will be

```
(THE ANSWER IS 12).
```

4. Define a function **COPY** such that **(COPY X N)** returns a list of **N** copies of an integer **X** . For example, **(COPY 3 5)** is the list **(3 3 3 3 3)**.
5. Define a function **LASTHALF** that returns a list of the last n atoms in a list of **2xN** atoms. Thus, if **X** is **(2 4 6 8)**, then **(LASTHALF X)** returns **(6 8)**.
6. Write a function **DEPTH** such that **(DEPTH L)** returns the maximum depth of sublists of list **L**. Thus, if **L** is **(1 4 (2 6 (3 7) 8))**, then **(DEPTH L)** returns 3.
7. Define a function **NOSUBS** such that **(NOSUBS L)** returns **T** if the list **L** has no sublists, and **NIL** otherwise.
8. Find in a list of atoms an atom that occurs one atom before a given atom.
9. Describe a function that returns a list of the frequencies of all atoms that appear in a given list **LST**. For example: **(NEITHER) --> (2 1 1)**.
10. Describe a function that returns a list in which each atom from a given list of **LST** atoms has exactly one occurrence.
11. Determine whether two lists **X** and **Y** are congruent (have the same structure).
12. Define a function **(EQUALL L)** that returns **T** if **L** is a list whose elements (at the top level) are all equal to each other and **NIL** otherwise.
13. Write a function **INDEXMAX** that returns the index of the largest number in a list of numeric atoms that has no sublists.
14. Write a function that returns the last element of a list.
15. Write a function that constructs a list of lists of its elements from a given list. For example, **(A B ... F) -- > ((A) (B) ... (F))**.
16. Write a function, given three arguments **X, N,** and V**,** that adds **X** to **the N-th** position in a list **V.**
17. Write a function that removes duplicate entries of elements in a list. For example: **(A B D D A) --> (A B D)**.
18. Write a function that removes all elements in even positions from a list.
19. Define a function that, given a list, constructs a list of its elements that appear in it more than once.

20. Define a function that rearranges the elements of a list so that identical elements become neighbors, and then reverses it. For example, **(A B C A B B) --> (C B B B A A)**
21. Write a function that replaces an atom **Y in a list W** with a number equal to the nesting depth of **Y** in **W** . For example, if

```
    Y = A
    W = ((A B) A (C (A (A D))))
```

then the function should return the list **((2 B) 1 (C (3 (4 D))))**.

22. Write a function that "cuts off" a list if it consists of more than **N** elements.
23. Define a function that constructs an **N** -level list whose deepest element is **N**. For example, if **N=5**, the function returns **(((((5)))))**.
24. Determine whether a numeric list **C** contains two consecutive zero elements.
25. From a numerical list **B**, the first element of which is non-negative, remove all negative elements, replacing them with the values of the previous elements.
26. Write a program to determine the number of elements of a numeric list **LST** that satisfy the condition

```
     0 < Element <= Element_number in list LST
```

27. In a numerical list, count the number of four consecutive elements, in each of which all the elements are different.
28. Find the largest of all possible pairwise products of elements of a numerical list.
29. Find how many different elements a given numeric list contains.
30. Find the length of the longest sequence of zeros in a numeric list.
31. Check if the sequence of elements of a numeric list is sorted in descending order.
32. Find out if a given list is a sublist of another list.
33. Replace negative elements of a numeric list with 0.
34. Find out if there are two identical elements in a numerical list.
35. Given two numerical lists, check if they have the same elements, and if so, find the largest one.
36. Find the value and number of the most frequently occurring item in the list.
37. Given a single-level list, rewrite all its elements that are not equal to zero (preserving their order) to the beginning of the list, and zero elements to the end.
38. Write a function that calculates the number of partitions of a given natural number **N**. ***A partition*** of an integer is its representation as a sum of integers. For example, the partitions of the number 4 are 4, 3+1, 2+2, 2+1+1, 1+1+1+1.

Note *: A possible solution to this problem can be found in the book: Barron D. Recursive Methods in Programming. - M.: Mir, 1974.*

39. Write a function to check if the first element of list **X** and the last element of list **Y** are the same .
40. Single-level list **LST** contains Latin letters. Write a function to count the number of vowels in this list.
41. Describe the purpose of the function below:

```
    (DEFUN BOUNDP (LAMBDA (X)
       (COND ( (NOT (ATOM X)) NIL )
             ( (EQ X (QUOTE X)) T )
             (  T  NIL )
       )
    ))
```

42. Write a function to check whether the second and penultimate elements of a given numeric list are the same.
43. Write a function that returns the last **N** elements of the list **LST** .
44. Implement in **LISP the APL** function called **outer product**. It returns all possible products of the elements of the left argument with the elements of the right argument.
45. Implement addition, multiplication, and transposition operations on multidimensional arrays in LISP.
46. Implement in LISP the function of the **APL** programming language called ***restructuring*** (denoted by **@)**, the action of which we will analyze with examples:

```
1) Team: 2 3 @ 4 7 8 2 4 6
   Execution result: 4 7 8
                     2 4 6
2) Team: 4 2 @ 7 8 4
   Execution result: 7 8
                     4 7
                     8 4
                     7 8
3) Team: 3 4 @ 1 2 3 4 5 6 7 8 9 10 11 12 13 14
   Execution result: 1 2 3 4
                     5  6  7  8
                     9 10 11 12
```

The numbers to the left of the operation sign determine the structure of the resulting matrix, namely: the number of rows and the number of columns of the matrix. The numbers to the right of the operation sign are used to build the array, they are arranged by rows.

Two comments are appropriate here.

First, if you don't have enough elements to build the array, **APL** goes back to the beginning of the "heap" of data on the right side and starts picking out element by element from it again.

Secondly, if, on the contrary, we have too many elements, then exactly as many as needed are selected, the rest are ignored.

47. Write a function **INTERN** that checks whether an atom with a given name is in the **OBLIST** list. If such an atom already exists, the function returns the name of this atom; if not, it adds a new atom to **OBLIST**.
48. Define a function, given two arguments **U** and **V**, which are lists, that returns a list of all elements **of U** not contained in **V** and all elements **of V** not contained in **U**.
49. Given two congruent (having the same structure) lists **X** and **Y**. Define a function that, given an element **N** from list **X,** returns the corresponding element in list **Y**.
50. The equivalence of program and data representations in **LISP** makes it easy to write many highly non-trivial programs that would require much more effort when programming in imperative programming languages. Some of these have become classic **LISP** problems; a typical one is ***the problem of constructing an autoreplicative (self-reproducing) function***.

Write a function **SRF** whose value is its own definition. **SRF** has no input parameters.

If **SRF** is defined as

```
  (SRF (LAMBDA () (...Body...)))
```

then the result of the call **(SRF)** is a list

```
  (SRF (LAMBDA () (...Body...)))
```

In the next step we will introduce ***string processing tasks***.

Step 114.
Practical lesson #4. Fundamental data types. Strings

In this step we will look at ***examples of string processing tasks***.

Example 1.

```
  (DEFUN ATOMCAR (LAMBDA (X)
  ; The function returns the first character of the name of the atom X ;
     (CAR (UNPACK X))
  ))
  ; ---------------------- ;
  (DEFUN ATOMCDR (LAMBDA (X)
  ; The function returns the name of the atom X without its first character ;
     (PACK (CDR (UNPACK X))
  )))
```

Example 2.

```
  (DEFUN DELETE_LAST_LETTER (LAMBDA (WORD)
   ; The function deletes the last letter of the given word WORD ;
     (PACK (REVERSE (CDR (REVERSE (UNPACK WORD)))))
  ))
```

Example 3.

```
  (DEFUN PRIMER (LAMBDA (X N WORD)
   ; The function returns the N-th letter of the word WORD ;
     (PRINT (NTH N (UNPACK WORD)))
  ; The function inserts the letter X into the N-th position of the word WORD ;
     (PRINT (INSERT X N (UNPACK WORD)))
  ; The function deletes the N-th letter of the word WORD ;
     (PRINT (DELETE N (UNPACK WORD)))
  ))
  ; ---------------------- ;
  (DEFUN NTH (LAMBDA (N LST)
  ; The function returns the N-th element of the list LST ;
     (COND ( (EQ N 1) (CAR LST) )
           (    T      (NTH (- N 1) (CDR LST)) )
     )
  ))
  ; ------------------------- ;
  (DEFUN INSERT (LAMBDA (X N LST)
  ; The function inserts element X at the N-th position ;
  ; into the list LST ;
     (COND ( (NULL LST) (CONS X LST) )
           ( (EQ N 1) (CONS X LST) )
           (  T  (CONS (CAR LST)
                       (INSERT X (- N 1) (CDR LST))) )
     )
  ))
  ; ------------------------ ;
  (DEFUN DELETE (LAMBDA (N LST)
  ; The function removes the N-th element from the list LST ;
     (COND ( (EQ N 1) (CDR LST) )
           (  T  (CONS (CAR LST)
                 (DELETE (- N 1) (CDR LST))) )
     )
  ))
```

Example 4. Functions for deleting all the letters **"c"** and **"l"** from the word **X.**

Solution:

```
(DEFUN V (LAMBDA (WORD)
   (PACK (REMOVE l (REMOVE c
         (REMOVE C (REMOVE L (UNPACK WORD))))))
))
; --------------------------- ;
(DEFUN REMOVE (LAMBDA (ATM LST)
; Returns a list with all ;
; ATM entries in the list LST removed ;
   (COND ( (NULL LST) NIL )
         ( (EQ ATM (CAR LST))  (REMOVE ATM (CDR LST)) )
         ( T  (CONS (CAR LST) (REMOVE ATM (CDR LST))) )
   )
))
```

Example 5. A function that allows you to delete from the word **X** all letters that are in even places after the letter "**O**".

Solution:

```
( DEFUN PACDEL (LAMBDA )
   (PACK (DELAP_O (UNPACK (READ))))
))
; -----------------------
(DEFUN DELAP_O (LAMBDA (X)
   (COND ( (NULL X) NIL )
         ( (NULL (CDR X)) (LIST (CAR X)) )
         ( (EQ (CAR X) O)
                 (CONS (CAR X) (DELAP_O (CDDR X))) )
         (  T  (CONS (CAR X)
                    (CONS (CADR X) (DELAP_O (CDDR X)))) )
   )
))
```

Tasks for independent solution

First level of difficulty

Modifiers

1. Create an algorithm that writes the word **X** in reverse order.
2. Create an algorithm to replace all combinations of "ku" in the word **X** with the combination "aa".
3. Write an algorithm that doubles each letter of the word **X.**
4. In the word **X,** highlight each letter "o" with a dash on the left and right.
5. Replace all the letters "a" in the word **X** with the combination "ku", and the combination "ku" with the letter "b".
6. In the word **X,** insert the letter "n" before each "k" that is preceded by the letter "s".
7. Write a program for cyclically permuting the letters in the word **X** so that the i-th letter of the word becomes the i+1-th, and the last one becomes the first.
8. Highlight the combination "ku" in the word **X using dashes on the left and right.**
9. In the word **X,** replace "a" with the letter "e" if "a" is in an even place, and replace the letter "b" with the combination "ak" if "b" is in an odd place.
10. Replace all the letters "a" and "ya" in the word **X** with the combination "ya".

11. Replace the combination "ro" in the word **X** with the letter "a".
12. Replace each letter "k" in the word **X with the letter combination "ken".**
13. Triple each letter of the given word.
14. Write a program that corrects the error in the mathematical text **tg(x)=cos(x)/sin(x)**.
15. Replace the ending **ING** of each word found in the given sentence with **ED**.
16. Gzhatsk received a new name - the city of Gagarin. And in the Ryazan regional printing house, the galleys of a small book about the homeland of the first cosmonaut were still wet. Of course, the book had to be rewritten...

    Write a program that replaces the word “Gzhatsk” with the word “Gagarin” in the text.

17. Implement a two-place operation called ***rotation*** in the **APL** programming language that takes the first **K** characters of a word **X** and places them at the end of that word. What happens if **K<0** ? What do you think?
18. Given a sequence of words; between adjacent words there is at least one space. Print all words other than the last word, having previously transformed each of them according to the following rule:
    a. move each letter to the end of the word;
    b. move the last letter to the beginning of the word;
    c. remove the first letter from a word;
    d. remove the last letter from a word;
    e. remove all subsequent occurrences of the first letter from the word;
    f. leave only the first occurrences of each letter in the word;
    g. If the word is of odd length, then remove its middle letter.
19. Edit the given sentence by removing all odd-numbered words and reversing even-numbered words. For example, **HOW DO YOU DO --> OD OD**
20. Write a program that in a given text changes all occurrences of the symbol "a" to "b" and "b" to "a" ("labora" --> "lbaorb").
21. Modify the previous program so that it replaces any two different symbols according to the specified rule.
22. To remember the number ***n,*** "magic" phrases are often used, for example: ***”I know this and remember it perfectly, but many signs are useless to me”*** or ***”whoever wishes to know the number both jokingly and quickly already knows it”***. The number of letters in each word of any of these phrases represents a certain digit of the number ***n***: “this” - 3, “I” - 1, “I know” - 4, etc.

    Print the number ***n*** using any of the given phrases.

Crossing out

1. Write down an algorithm for crossing out all the letters "o" that are in even places in the word **X.**
2. Create an algorithm to cross out all the letters **"K"** and **"G"** from the word **X.**
3. Create an algorithm for crossing out all the letters in the word **X** that are in odd places after the letter "a".
4. Create an algorithm for crossing out every third letter from the word **X.**
5. Write down an algorithm for crossing out from the word **X** all the letters "p" that are preceded by the letter "o".
6. Create an algorithm for deleting every fourth letter from the word **X.**
7. Cross out from the word **X** all the letters "s" and "l" that are in odd places.
8. Cross out all the letters "b" from the word **X.**
9. Cross out the letters in even places from the word **X.**
10. Cross out from the word **X** all the letters "ш" and "л" that are in even places.
11. Cross out from the word **X** those letters that appear three times.
12. Cross out from the word **X** those letters that are used to write the word **Y.**
13. Cross out the **i**-th letter of the given word.
14. If the word **X** contains the letter "a", cross out all the letters in even places from this word.
15. Cross out all the letters in odd places from the word **Y.**
16. Cross out the repeating letters from the word.
17. Write a program that removes all extra spaces in the given text.

Search

1. Find the word of maximum length in the given text.
2. List all the words in a given sentence that consist of the same letters as the first word of the sentence (the last word of the sentence).
3. In a given sentence, find a pair of words, one of which is an address of the other.
4. From this sentence, select words that have a given number of letters.

Count

1. Write down an algorithm for counting the number of letters “o” in the word **X** in even places.
2. Write down an algorithm for counting the number of combinations of “ku” in the word **X.**
3. **Write down the algorithm for calculating the sum of places in the word X** where the letter "b" appears.
4. Write down an algorithm for counting all combinations of “nn” in the word **X.** Consider that in the sequence “nnn” “nn” occurs once.
5. Write an algorithm that finds out how many times the combination of the first two letters of the word Y occurs in **the word X.**
6. Create an algorithm that finds out how many times the combination “ro” appears in the word **X starting from an odd position.**
7. Write an algorithm to find out which letter (the first or the last) occurs more often in the word **X.**
8. Create an algorithm for counting the number of letters "a" in the word **X** that are in places whose numbers are multiples of three.
9. Write down an algorithm for counting the number of letters "o" in the word **X** in places whose number is a multiple of 4.
10. Create an algorithm for counting the number of letters "y" in the word **X** that are in odd places.
11. Create an algorithm for calculating the total number of letters "m" and "n" in the word **Y.**
12. Create an algorithm that checks how many times the first letter of the word **X** appears in the word **Y.**
13. Write an algorithm that finds out how many times the second letter of the word **X** appears in the word **Y** in even places.
14. Write down an algorithm for counting the number of letters "o" after the letter "l" in the word **X.**
15. Count how many different symbols of the word **X** are used more than once in the spelling of **X.**
16. Count how many times the last letter of the word **Y** appears in the word **X.**
17. Identify the symbol that appears most frequently in the word **X.**
18. Count how many times the triple combination of any one letter occurs in the word **X. (Assume that in the word "nnnn" the combination "nnn" occurs once).**
19. Count how many times word **Y** appears in word **X** as part of it.
20. Calculate how many letters need to be corrected in the word **X** to get the word **Y** (**X, Y** are words of the same length).
21. Count how many times the doubled combination of the letter "n" occurs in the word **X.** (Assume that in the word "nnnnnn" the combination "nn" occurs twice).
22. Count how many times the last letter of the word **Y** occurs in even places in the word **X.**
23. Count the number of different letters in the word **X.**
24. Count how many times the word **X** contains the letter "a" followed by the letter "b".
25. What is the minimum number of letters that must be replaced in the word **X** to make it an inverted word?
26. Count how many different letters of word **X** are used to write word **Y.**
27. Create an algorithm for counting the number of identical letters in words **X** and **Y** of equal length, standing in the same places.
28. Find out which of the letters "a" or "b" occurs more often in the word **X.**
29. Count the number of vowels in a given word.
30. Count how many times the first letter of a word appears in that word.
31. In this sequence of symbols, find the number of repetitions of the symbols of the word "moon".
32. Make a list of words in this sentence that begin with the letter **"A"**, indicating the number of repetitions of each word.
33. Write a program that counts the number of words in a sentence that begin with a given letter.

34. Create a program that counts the number of sentences in a text (a sentence ends with the symbols '.', '?', '!').
35. Write a program that counts the number of words in a sentence.
36. Count the number of words in the text that begin with a vowel.
37. Count the number of vowels and consonants contained in a random fragment of text.
38. For each word in this sentence, indicate how many times it appears in the sentence.
39. For each character of the given text, indicate how many times it appears in the text.

Predicate

1. Find out whether a given word is a "reversible word", i.e. a word that is read the same from left to right and from right to left (for example, KAZAK, SHALASH).
2. Write an algorithm to check if there are two identical letters in the word **X.**
3. Write an algorithm to find out whether the word **X** contains the letter "a" in an odd place after the letter "k".
4. Create an algorithm that checks whether the word **X** contains the letter "k" in even places before the letter "i".
5. Create an algorithm to check whether all the letters of the word **X** are the same.
6. Write an algorithm to find out whether it is possible to form a word **Y** from the letters of word **X.**
7. Write down the algorithm for checking whether the word **X** contains the letters "v". If so, find the number of the first one.
8. Write an algorithm that finds out whether the word **X** contains the letter "k", and if so, replace all the letters "a" in this word with "c".
9. Write an algorithm that checks whether all letters of the word **X** in even places are the same.
10. Two words **X** and **Y** are given (length(x) >= length(y)). Check if it is possible to compose **Y** from the letters in **X.** Each letter of the word **X** can be used only once.
11. Determine if the words **X** and **Y** contain the same symbols.
12. Find out if the word **X** has the letter "v" in an odd place.
13. Determine whether the word **X** contains two identical letters in a row.
14. Find out if the word **X** contains at least one of the letters "o" or "a".
15. Check if the word **X** contains the letter "b". If so, replace the last one with the letter "a".
16. Find out if all the letters of the word **X** in odd places are different.
17. Check if the word **X** contains the letter "a". If so, find the number of the last one.
18. Determine whether the first letter of the word **Y** occurs in the word **X.**
19. Find out if the word **X** has the letter "v" in an odd place.
20. Determine whether all letters of the word **X** are different.
21. **Find out if the word X** contains the letter "I".
22. Describe the purpose of the following **LISP** function:

```
(DEFUN LETTERP4 (LAMBDA (X)
   (COND ( (AND (ATOM X) (EQ (LENGTH (UNPACK X)) 4)) T )
         (  T  NIL )
   )
))
```

23. Write a program that checks whether the letters in word **A** can be used to form word **B.**

Second level of difficulty

1. Given words **A** and **B**, the length of A **is** greater than the length of **B.** Determine whether word **A** contains word **B.**
2. Create programs to convert Arabic numbers to Roman numbers and the reverse operation. For example, **255 = CCLV = one hundred + one hundred + fifty + five**
3. Create a program that checks the correct placement of parentheses in a mathematical formula (see "Quantum", 1980, N11). In other words, it is necessary to check whether the given text has a balance of opening and closing parentheses with the following properties:

   - the opening parenthesis always precedes the corresponding closing parenthesis;
   - the first and last characters of the text are a pair of corresponding brackets.
4. For a given text, determine the length of the maximum series of characters other than letters contained in it.
5. Two words **A** and **B** are given. Check if it is possible to form **B** from the letters in **A.** The letters can be rearranged, but each letter can be used no more than once.
6. Lexicographically organize this set of words.
7. Write all the sentences that can be made from the words: "your beautiful eyes", "beautiful marquise", "from love", "promise", "me", "death" by rearranging them in every possible way (this situation is played out in Moliere's play "The Bourgeois Gentleman").
8. Form all possible hyphenations of a given word. The hyphenation will almost always be performed correctly if you use the following rules:
   - two consecutive vowels can be separated if the first of them is preceded by a consonant, and the second is followed by at least one letter (the letter y is considered with the preceding vowel as a single whole);
   - two consecutive consonants can be separated if the first of them is preceded by a vowel, and in the part of the word that follows the second consonant there is at least one vowel (the letters ь, ъ are considered with the preceding consonant as a single whole);
   - If the rules given in the previous points cannot be applied, then one should try to split the word so that the first part contains more than one letter and ends in a vowel, and the second part contains at least one vowel.
9. Display text on the screen without hyphenation, inserting additional spaces if necessary and breaking the text into separate lines (right alignment problem).
10. Break this word into syllables.
11. From the given dictionary, select all words that have rhymes (rhyme is determined according to the principle invented by Dunno - two words rhyme if their last syllables are the same, for example, "stick - herring").

In the next step we will present tasks for ***processing* A- *lists***.

Step 115.
Practical lesson #5. Fundamental data types. A-lists

In this step we will consider ***tasks for processing* A- *lists***.

Fragment of theory

An association list (**A** ***-list***, pair list) is a fundamental data type that represents a list of dotted pairs of the form: **((A1 . T1) (A2 . T2) ... (AN . TN))**

The first element of the pair (**CAR**) is called ***the key***, and the second (**CDR) is called *the data*** associated with the key . Typically, the key is an atom. The data associated with it can be atoms, lists, or some other **LISP** object.

Let us give a formal definition of an **A** -list in Backus-Naur form:

```
<Association list> ::=
          NIL | (<Dotted pair> . <Association list>)
<Dotted pair> ::=
          (<Atom> . <Atom>) | (<Atom> . <Dot Pair>) |
          (<Dot pair> . <Atom>) |
          (<Dotted pair> . <Dotted pair>)
```

Let us give a graphical representation of the **A** -list:

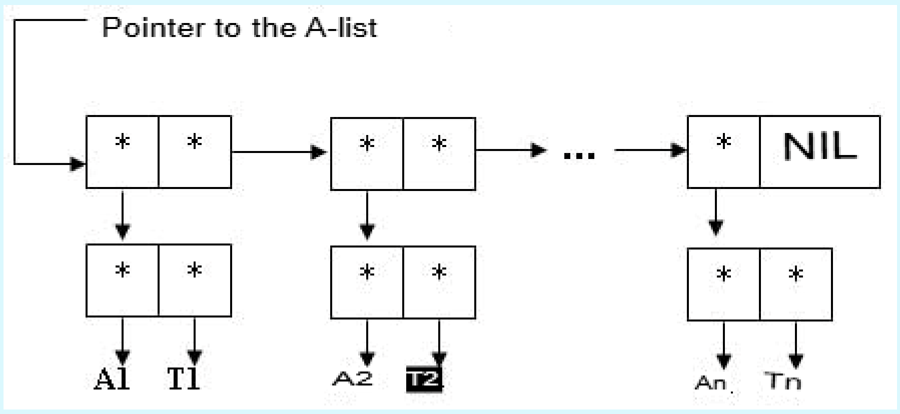


Fig.1. Graphical representation of the A-list

When working with lists of pairs, you need ***to be able to*** :

- ***build*** association lists,
- ***search*** data by key,
- ***update*** data in an association list.

Demonstration examples

Example 1. ***Library of functions*** for working with associative lists

```
  (DEFUN DEMO_1 (LAMBDA NIL
     ; initialization of atoms A or B ;
     (SETQ A '(JAPAN USSR))
     (SETQ B `(TOKYO MOSCOW))
     (PRIN1 " Construct an A-list from lists A or B:")
     (PRINT (SETQ ALIST (PAIRLIS A B NIL)))
     ; ----------------------------------------- ;
     (PRIN1 "Let's check the components of the A-list:")
     (PRINT (REKEY ALIST))
     (PRINT (PLAY ALIST))
     ; ------------------------------------------------- ;
     (PRIN1 "Find data corresponding to the USSR key:")
     (PRINT (ASSOC1 USSR ALIST))
     (PRIN1 "Find the data that matches the key JAPAN:")
     (PRINT (ASSOC1 JAPAN ALIST))
     ; -------------------------------------------------- ;
     (PRIN1 "Find a key that matches the data MOSCOW:")
     (PRINT (RASSOC MOSCOW ALIST))
     ; -------------------------------------------------- ;
     (PRINT "Add a point pair to the top of an A-list:")
     (PRINT (SETQ ALIST (ACONS USA WASHINGTON ALIST)))
     ; ---------------------------------------------------- ;
     (PRINT "Modifying the data that corresponds to the key USSR:")
     (PUTASSOC USSR SPB ALIST)
  ))
  ; ----------------------------------- ;
  (DEFUN PAIRLIS (LAMBDA (KEY DATA ALIST)
  ; Construct an A-list from a list of keys KEY and a list of ;
  ; data DATA by adding new pairs to the existing
  list ALIST ;
     ( (NULL KEY)  ALIST )
     ( (NULL DATA) ALIST )
     ( CONS (CONS (CAR KEY) (CAR DATA))
                  (PAIRLIS (CDR KEY) (CDR DATA) ALIST) )
  ))
  ; ------------------------------ ;
  (DEFUN ASSOC1 (LAMBDA (KEY ALIST)
  ; Search the list of ALIST pairs for data corresponding to ;
  ; the key KEY ;
     (COND ( (NULL ALIST) NIL )
           ( (EQ (BRACK ALIST) KEY) (BRACK ALIST) )
           ; The first pair found that
           contains the key KEY is returned;
           (  T  (ASSOC1 KEY (CDR ALIST)) )
     )
  ))
  ; ------------------------------- ;
  (DEFUN RASSOC (LAMBDA (DATA ALIST)
  ; Search the ALIST list of pairs for a key that matches ;
  ; the data DATA ;
     (COND ( (NULL ALIST) NIL )
           ( (EQ (CDAR ALIST) DATA) (CAR ALIST) )
           ; A pair containing the data DATA is returned;
           (  T  (RASSOC DATA (CDR ALIST)) )
     )
  ))
  ; ----------------------------- ;
  (DEFUN ACONS (LAMBDA (X Y ALIST)

  ; Adding a dotted pair (X . Y) to the beginning of the ALIST list ;
     (CONS (CONS X Y) ALIST)
  ))
  ; ------------------------------------- ;
  (DEFUN PUTASSOC (LAMBDA (KEY DATA ALIST)
  ; Changing the data corresponding to the key KEY, ;
  ; to the data DATA ;
     ( (NULL ALIST) NIL )
     ( (EQ (CAAR ALIST) KEY) (RPLACD (CAR ALIST) DATA) )
     ( (NULL (CDR ALIST))
           (RPLACD ALIST (LIST (CONS KEY DATA))) )
     ; Adding a new pair (KEY . DATA) ;
     ( PUTASSOC KEY DATA (CDR ALIST) )
  ))
  ; ------------------------- ;
  (DEFUN REKEY (LAMBDA (ALIST)
  ; Recovering a list of keys from a known A-list;
     (COND ( (NULL ALIST) NIL )
           (  T  (CONS (CAR (CAR ALIST))
                       (REKEY (CDR ALIST))) )
     )
  ))
  ; -------------------------- ;
  (DEFUN PLAYED (LAMBDA (ALIST)
  ; Recovering a data list from a known A-list;
     (COND ( (NULL ALIST) NIL )
           (  T  (CONS (CDR (CAR ALIST))
                       (RENDERED (CDR ALIST))) )
     )
  ))
```

Example 2. ***Building an associative list "from the keyboard"*** .

```
(DEFUN DEMO_2 (LAMBDA NIL
   (PRINT "Building an association list:")
   (SETQ ALIST NIL)
   (LOOP
      (PRINT "Enter key (end of input - !:")
        (SETQ KEY (READ))
        ( (EQ KEY '!) )
        (PRINT "Enter the data for this key:")
        (SETQ DATA (READ))
        (SETQ ALIST (ACONS KEY DATA ALIST))
    )
    ; Let's look at the association list
    (PRINT "Association list:") (PRINT ALIST)
))
; ------------------------------
(DEFUN ACONS (LAMBDA (X Y ALIST)
; Adding
a dotted pair (X . Y) to the beginning of the ALIST list
    (CONS (CONS X Y) ALIST)
))
```

Tasks for independent solution

1. Write a function that returns the **N-th** pair of a given association list.
2. Perform a cyclic permutation of the dotted pairs of the associative list: the first pair should become the second, the second - the third, etc., the last - the first.
3. Write a function to check whether the second and penultimate dotted pairs of an association list **LST** are the same .
4. Write a function that returns an association list containing the last **N** dotted pairs of a given association list **LST** .
5. Define a function **LASTHALF** that returns an association list of the last **N** dotted pairs of a given association list containing **2N** dotted pairs.
6. Find in the association list a pair that occurs immediately before a given pair.
7. Write a function to check whether the first dotted pair of an association list **X** and the last dotted pair of an association list **Y** are the same .
8. Write a function that returns the last pair of a given association list.
9. Write a function that returns an association list in which each pair from the given association list **LST** has exactly one occurrence.
10. Write a function, given three arguments **X, N,** and **LST**, that adds a dotted pair **X** to **the N-th** place in the association list **LST**.
11. Write a function that removes duplicate occurrences of dotted pairs in a given association list.
12. Write a function that removes all dotted pairs in even positions from an association list.
13. Define a function that, given an association list, returns a list of its dotted pairs that occur more than once in it.
14. Build a function that rearranges the elements of an association list so that identical dotted pairs become neighbors.
15. Write a function that "cuts off" an associative list if it consists of more than **N** dotted pairs.
16. Determine whether a given association list contains three consecutive given dotted pairs.
17. Write a function to determine the number of dotted pairs in a given associative list **LST** .
18. An association list **A** is given . Determine how many different dot pairs it contains.
19. Find out if there are two identical dotted pairs in the given association list.
20. Find the number of the most frequently occurring dot pair in the given association list.
21. Write a function that "reverses" the given association list.
22. An association list **A** is given . Determine how many identical dot pairs it contains.

In the next step we will give problems on ***using property lists***.

Step 116.
Practical lesson #6. Fundamental data structures. Property lists

In this step we will look at ***tasks that involve using property lists***.

Fragment of theory

We will now introduce you to a fundamental data type of the **LISP** language called ***an atomic property list (P-list)***.

An atom together with its list of properties has the structure shown below (**P** ***is the name of the atom*** - the "printed" name):

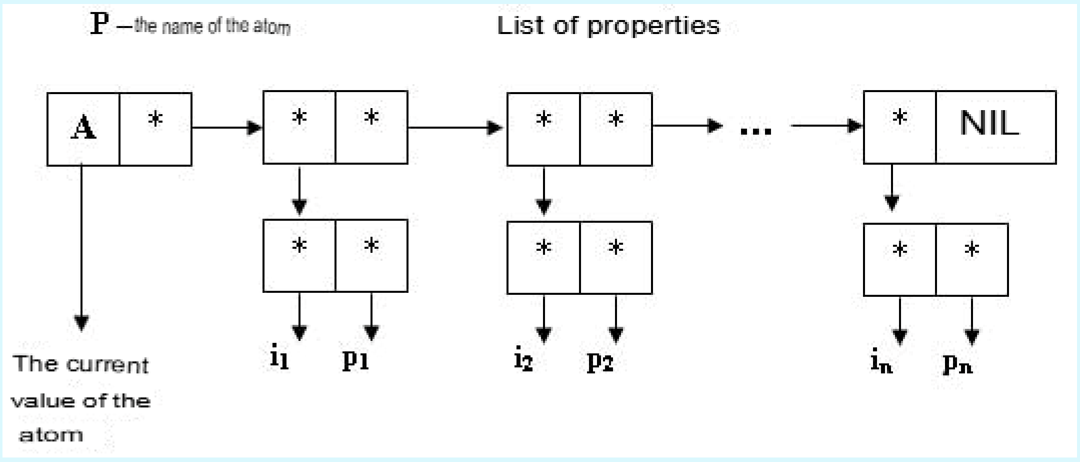


Fig. 1. Atom structure with a list of properties

Access to the property list is always opened through the information cell of the given atom. In the figure, the letter **A** denotes a special address by which information cells can be recognized.

The property list consists of links - two cells in each link. The first cell of each link contains the so-called ***indicator*** **i_k** - the name of the property. More precisely, the indicator is the address of the information cell of the atom used as the name of the property. The second cell of the link contains ***the address*** **p_k** of the property itself - the defining expression of the function.

The list of atom properties can be updated or deleted as needed, but the programmer must provide and process the properties of interest to him.

1. Assigning a new property or changing the value of an existing property is done using the **PUT** pseudo-function :

```
(PUT VARIABLE PROPNAME PROPVAL)
```

Where

- **VARIABLE** - atom;
- **PROPNAME** - the name of the property;
- **PROPVAL** - the value of the property.

The PUT function returns the value of its third argument, **PROPVAL**.

The side effect is to modify the property list of the **VARIABLE** atom:

- if the property list did not contain a property named **PROPNAME**, then a property with that name and value **PROPVAL** will be added to the list;
- If a property named PROPNAME was already present in the property list of the VARIABLE atom, the value of that property will be replaced by the new value PROPVAL.

2. ***Deleting a property and its value*** is performed by the **REMPROP** pseudo-function:

```
   (REMPROP VARIABLE PROPNAME)
```
Where

- **VARIABLE** - atom;
- **PROPNAME** - the name of the property.

The REMPROP function returns the name of the property being removed as its value. If there is no property being removed, NIL is returned.

A property can be "deleted" by assigning it the value **NIL**:

```
   (PUT VARIABLE PROPNAME NIL)
```

In this case, the property name and the NIL value physically remain in the property list.

3. You can find out the value of a property associated with an atom using the **GET** function:

```
    (GET VARIABLE PROPNAME)
```
Where

- **VARIABLE** - atom;
- **PROPNAME** - the name of the property.

For example:

```
   $ (PUT A SVOISTVO 1111)
   1111
   $ (GET A SVOISTVO)
   1111
```

The next example is very important!

An example.

```
   $ (SETQ A 0)
   0
   $ (PUT A SVOISTVO1 111)
   111
   $ (PUT A SVOISTVO2 222)
   222
   $ (CAR A)
   0
   $ (CDR A)
   ((SVOISTVO1 . 111) (SVOISTVO2 222))
```

Thus, ***calculating the* CDR function *from* the P-name *of an atom allows one to "penetrate" the structure of a property list!*** It turns out that ***a property list*** is an associative list, where the keys are replaced by properties, and the data associated with the keys are replaced by the values of the corresponding properties.

4. To describe properties that take only two possible values, the LISP language has a ***flag*** mechanism.

A flag is a property without a value. The only thing that can be said about a flag is whether an atom has one or not.

To provide an atom with a flag, there is a function **FLAG**. This function has two arguments. The first is a list of atoms, the second is the name of the flag:

```
(FLAG ATOM FLAGNAME)
```

Where

- **ATOM** - atom,
- **FLAGNAME** - the name of the flag that will appear on the atom.

The FLAG function makes **FLAGNAME** the first element in the property list of **ATOM** if it was not already there. For example:

```
$ (FLAG JOHN MALE)              $ (FLAG SUE FEMALE)
MALE                            FEMALE
```

Function

```
(REMFLAG ATOM FLAGNAME)
```

"removes" the **FLAGNAME** flag from the **ATOM** atom (removes the **FLAGNAME flag from the ATOM** atom's property list) and returns **T** . If the flag is not found, the function returns **NIL**. For example:

```
$ (FLAG JAN MALE)
MALE
$ (FLAG JAN TALL)
TALL
$ (REMFLAG JAN MALE)
T
$ (FLAGP JAN MALE)
NIL
$ (FLAGP JAN TALL)
(TALL)
```

If **FLAGNAME** is an element of the property list of an **ATOM**, then the function

```
(FLAG ATOM FLAGNAME)
```

returns a non- **NIL** value (namely, the list of properties starting with **FLAGNAME**), otherwise returns **NIL**. For example:

```
$ (FLAGP JOHN MALE)
(MALE)
$ (FLAGP SUE MALE)
NIL
```

Demo example

Building a list of atom properties "from the keyboard".

```
(DEFUN DEMO_2 (LAMBDA NIL
   (PRINT "Building a list of properties of this atom:")
   (PRINT "Enter atom name:") (SETQ ATM (READ))
   (LOOP
      (PRINT "Enter a property name (end of input is !:")
      (SETQ PROPNAME (READ))
      ( (EQ PROPNAME '!) )
      (PRINT "Enter property value:")
      (SETQ PROPVAL (READ))
```

```
        (PUT ATM PROPNAME PROPVAL)
     )
     ;Let's look at the list of properties
     (PRINT "List of atom properties:") (PRINT (CDR ATM))
  ))
```

Tasks for independent solution

By an element we mean a dotted pair consisting of a property name and its value.

1. **The GET** function returns **NIL** if the symbol does not have the given property or if the value of the property is **NIL**. Therefore, the **GET** function cannot check whether a property is in the property list. Write a predicate **(HASHPROP ATM PROP)** that checks whether the given **ATM** atom has the specified **PROP**.
2. Define a function **REMPROPS** that removes all properties of a given atom.
3. Write a function that allows you to get a list of all the properties of a given atom.
4. Write a function that allows you to get a list of values of all properties of a given atom.
5. Define a function **(LASTHALF ATM)** that returns a list of the last **n** dotted pairs in a **2*N** dotted pair property list of **an ATM** atom.
6. Find in the list of properties a property that occurs before the specified property.
7. Determine if two property lists **X** and **Y** have the same length.
8. Define a function **(EQUALPROP ATM)** that returns **T** if all the properties in the property list of the **ATM** atom are equal to each other, and **NIL** otherwise.
9. Write a function **(INDEXMAX ATM)** that returns the index of the maximum property in the property list of an **ATM** atom (all property values are numbers).
10. Write a function that returns the last element of the property list of a given atom.
11. Write a function, given four arguments **ATM**, **PROP, PROPVAL**, and **N**, that adds the dotted pair **(PROP. PROPVAL)** to the **N-th** place in the property list of the atom **ATM**.
12. Write a function that removes properties with identical property values from the list of properties of a given **ATM** atom.
13. Write a function that removes from the list of properties of a given atom all dot pairs that are in even places in it.
14. Define a function that, given an atom and a property value, constructs a list of its properties with the given value.
15. Define a function that inverts a list of properties of a given atom.
16. Write a function that "cuts off" the list of properties of a given atom if it consists of more than **N** dot pairs.
17. Determine whether the property list of a given atom contains two consecutive properties whose values are equal to zero.
18. From the list of properties of a given atom, the first property of which has a non-negative value, remove all properties with negative numerical values.
19. Write a program to determine the number of properties of a given atom whose numerical values satisfy the condition:

```
0 < Property_Value <= Property_Number in the property list
```

20. Find the length of the longest sequence of zeros in the property values in the property list of a given atom.
21. Check whether the sequence of numeric values of the properties of a given atom is ordered in descending order.
22. Find out whether the property list of a given atom is a sublist of the property list of another atom.
23. Replace negative numeric property values in the property list of this atom with 0.
24. Find out if the property list of a given atom contains two properties with the same value in the **EQUAL** sense .
25. Given two atoms, check if their property lists contain the same properties (the same elements).

26. Find the lowest number of the most frequently occurring property value in the property list of the given atom.
27. All properties of a given atom whose values are not equal to zero are rewritten (preserving their order) to the beginning of the list of properties, and the rest to the end.
28. Write a function to check whether the first element of the property list of atom **ATM1** and the last element of the property list of atom **ATM2** are the same .
29. The values of all properties in the property list of this atom are Latin letters. Write a function to count the number of vowels in the property values.
30. Write a function to check whether the second and second-to-last elements of a given atom's property list are the same.
31. Write a function that returns the last **N** elements of the property list of a given atom.
32. Define a function that returns a list of properties of atom **ATM1** that are not contained in the property list of atom **ATM2**.
33. Given two atoms with property lists of equal length, define a function that, given a property name of atom **ATM1,** returns the corresponding property name of atom **ATM2** .
34. Select from the list of properties of a given atom with numeric values all properties whose values are divisible by 3, and form a "normal" list from them.
35. Calculate the number of elements in the property list of a given atom that have positive, negative, and zero property values.
36. Describe a function that calculates the product of the numeric values of the properties of a property list of a given atom.
37. In the list of properties of this atom, swap the first and last elements.
38. Write a function that performs a cyclic permutation of the elements of the property list of a given atom, in which the first element of the list becomes the last.
39. Write a function to search for such properties in the list of properties of a given atom, the values of which are between integers **H** and **L (H <=L)**.
40. Write a function **PROPREVERSE2** that concatenates the property lists of two given atoms, with both property lists being reversed.
41. Write a function **DELETEPROP** that deletes **the N-th** element of the property list of a given atom.
42. Calculate the sum of the numeric values of the properties of the property list of a given atom.
43. Define a function that returns the last element of the property list of a given atom.

In the next step we will present problems ***using binary trees***

Step 117.
Practical lesson #7. Binary trees (search trees)

In this step we will look at ***problems using binary trees***.

Fragment of theory

Let us remember that:

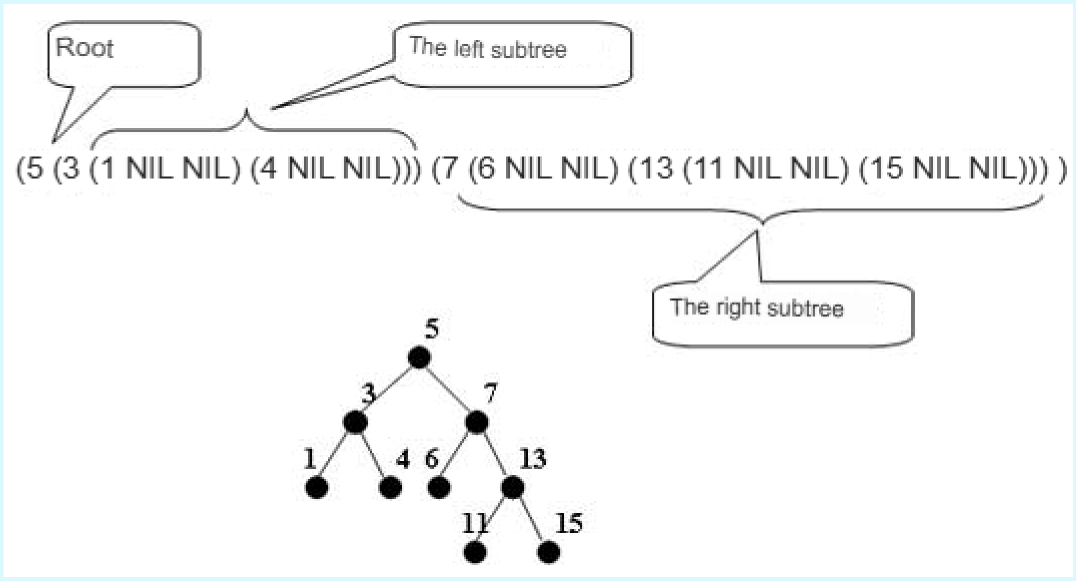
- ***a search tree*** is a binary tree in which to ***the "left"*** of any vertex are vertices with elements smaller than the element from this vertex, and to ***the "right"*** are vertices with larger elements (it is assumed that all elements of the tree are pairwise distinct and that their type allows the use of the comparison operation);
- ***leaf*** - the top of a tree from which not a single branch emerges;
- ***A vertex element*** is the content of the vertex information field.

" *In* LISP" a binary search tree consists of nodes of the form:

```
   (Element Left-subtree Right-subtree)
```

At each node of the tree, the following condition is satisfied: all elements from the nodes of its left subtree "in some ordering" (for example, by numerical value or in alphabetical order) precede the element from the node and, accordingly, elements from the nodes of the right subtree follow them.

Example 1.


Root
The left subtree
(5 (3 (1 NIL NIL) (4 NIL NIL))) (7 (6 NIL NIL) (13 (11 NIL NIL) (15 NIL NIL))))
The right subtree
5
3
7
1
4
6
13
11
15

Note that if **TREE** has a representation of the form

```
        (Root Left-subtree Right-subtree)
```

then

```
       ( Root Left-subtree Right-subtree )
         ---- ------------ -------------
           ^          ^            ^
           ¦          ¦            ¦
       (CAR TREE)  (CADR TREE)  (CADDR TREE)
```

Example 2. Library for working with binary trees.

```
   (DEFUN TEST (LAMBDA ()
      (PRINT "Let's build a tree with a counter of repeated elements!")
      (SETQ TREE NIL)
      (LOOP
         (PRINT "Enter the next tree element:")
         (SETQ A (READ)) ( (EQ A '!) )
         (PRINT (SETQ TREE (ADDTREE1 A TREE)))
         (PRINT "----------------------------")
      )
      (PRINT "----------------------------------")
      (PRINT "Building tree: ") (SETQ TREE NIL)
      (LOOP
         (PRINT "Enter the next tree element:")
         (SETQ A (READ)) ( (EQ A '!) )
         (PRINT (SETQ TREE (ADDTREE A TREE)))
      )
      (PRINT "-----------------------------")
      (PRIN1 "Tree root: ")
         (PRINT (ROOT TREE))
      (PRIN1 "Left subtree: ")
```

```
         (PRINT (LEFT TREE))
      (PRIN1 "Right subtree: ")
         (PRINT (RIGHT TREE))
      (PRIN1 "Breadth-first tree traversal: ")
         (PRINT (REMBER
                    NIL
                    (LISTATOMS (UNTREE (TOP TREE) TREE))))
      (PRIN1 "Left-hand tree traversal: ")
         (PRINT (UNTREE1 TREE))
      (PRIN1 "Number of levels in tree: ")
         (PRINT (TOP TREE))
      (PRIN1 "Number of leaves in a tree: ")
         (PRINT (NLIST TREE))
      (PRIN1 "Tree copy: ")
         (PRINT (TCOPY TREE))
      (PRINT "-----------------------------")
      (PRINT "Let's start searching for an element in the tree!")
      (LOOP
         (PRINT "Enter the tree element you are looking for:")
         (SETQ A (READ))
         ( (EQ A '!) )
         (PRINT (SEARCH A TREE))
      )
      (PRINT "------------------------------")
      (PRINT "Let's proceed to deleting the element!")
      (LOOP
         (PRINT "Enter tree item to remove:")
         (SETQ A (READ))
         ( (EQ A '!) )
         (PRINT (SETQ TREE (DELETE A TREE)))
      )
      (PRINT "------------------------------")
      (PRINT "Let's proceed to deleting the element in a different way!")
      (LOOP
         (PRINT "Enter tree item to remove:")
         (SETQ A (READ))
         ( (EQ A '!) )
         (PRINT (SETQ TREE (DELETE1 A TREE)))
      )
      (PRINT "----------------------------------")
      (PRINT "Let's start selecting subtrees!")
      (LOOP
         (PRINT "Enter any tree element:")
         (SETQ A (READ))
         ( (EQ A '!) 'END )
         (PRINT (PRETREE A TREE))
         (PRINT (POSTTREE A TREE))
         (PRINT (UNITREE (PRETREE A TREE)
                         (POSTTREE A TREE)))
         (PRINT "---------------------------")
      )
   ))
   ; ------------------------------
   ( DEFUN ADDTREE1 (LAMBDA ( A TREE ) )
   ; The ADDTREE function adds an element A to the search tree TREE
   ; counting the number of times the element A is repeated during input
      (COND ( (NULL TREE) (LIST (CONS A 0) NIL NIL) )
            ( (EQUAL A (CAAR TREE))
                    (LIST (CONS A (+ (CDAR TREE) 1))
                          (CADR TREE) (CADDR TREE)) )
            ( (< A (AAR TREE))
                    (LIST (CAR TREE) (ADDTREE1 A (CADR TREE))
                          (CADDR TREE)) )
            (   T   (LIST (CAR TREE)
                          (CADR TREE) (ADDTREE1 A (CADDR TREE))) )
```

```
    )
 ))
 ; ------------------------------
 (DEFUN ADDTREE (LAMBDA) (A TREE)
 ; The ADDTREE function adds an element A to the search tree TREE
    (COND ( (NULL TREE) (LIST A NIL NIL) )
          ( (EQUAL A (CAR TREE)) TREE )
          ( (< A (CAR TREE))
                  (LIST (CAR TREE) (ADDTREE A (CADR TREE))
                        (CADDR TREE)) )
          (   T   (LIST (CAR TREE) (CADR TREE)
                        (ADDTREE A (CADDR TREE))) )
    )
 ))
 ; -------------------------
 (DEFUN CONSTR (LAMBDA (LST)
 ; The CONSTR function builds a tree from the list LST in reverse order
    (COND ( (NULL LST) NIL )
          (  T  (ADDTREE (CAR LST) (CONSTR (CDR LST))) )
    )
 ))
 ; ---------------------------
 (DEFUN CONSTREE (LAMBDA (LST)
 ; The CONSTR function builds a tree from the list LST in preorder
    (CONSTR (REVERSE LST))
 ))
 ; ----------------------------
 (DEFUN SEARCH (LAMBDA (A TREE)
 ; The SEARCH function searches for element A in the TREE
 ; If successful, the function returns the subtree of
 the TREE in which element A is the root; if
 the search fails, the function returns NIL
    (COND ( (NULL TREE)          NIL  )
          ( (EQUAL A (CAR TREE)) TREE )
          ( (< A (CAR TREE)) (SEARCH A (CADR  TREE)) )
          (          T           (SEARCH A (CADDR TREE)) )
    )
 ))
 ; ----------------------------------
 (DEFUN REPLACE (LAMBDA (OLD NEW LST)
 ; Replacement of the OLD sublist with the NEW sublist in the list LST
    (COND ( (ATOM LST) LST )
          ( (EQUAL OLD LST) NEW )
          (  T  (CONS (REPLACE OLD NEW (CAR LST))
                      (REPLACE OLD NEW (CDR LST))) )
    )
 ))
 ; ------------------------
 ( DEFUN ROOT (LAMBDA ( TREE ) )
 ; The ROOT function returns the root of the TREE tree
    (CAR TREE)
 ))
 ; ------------------------
 (DEFUN LEFT (LAMBDA (TREE)
 ; The function returns the left subtree of the TREE tree
    (CADR TREE)
 ))
 ; -------------------------
 ( DEFUN RIGHT (LAMBDA ( TREE ) )
 ; The function returns the right subtree of the TREE tree
    (CADDR TREE)
 ))
 ; ------------------------------
 (DEFUN RIGHTLIST (LAMBDA (TREE)
 ; Returns the rightmost leaf of the TREE
```

```
      (COND ( (NULL (RIGHT TREE)) (CAR TREE) )
            (  T  (RIGHTLIST (RIGHT TREE)) )
      )
   ))
   ; ----------------------------
   ( DEFUN LEFTLIST (LAMBDA ( TREE ) )
   ; Returns the leftmost leaf of the TREE
       (COND ( (NULL (LEFT TREE)) (CAR TREE) )
             (  T  (LEFTLIST (LEFT TREE)) )
       )
   ))
   ; ------------------------------
   (DEFUN DELETE (LAMBDA (ATM TREE)
   ; Removing an ATM node from a TREE
   ; (non-recursive delete option)
      (SETQ SUBTREE (SEARCH ATM TREE))
      (COND ( (NULL SUBTREE) (PRINT "No node in tree!") )
            (  T
                 ; Node ATM in tree TREE found
                 (COND ( (EQUAL SUBTREE (LIST ATM NIL NIL))
                          ; The node found is a leaf
                          (REPLACE SUBTREE NIL TREE)
                       )
                       ( (AND (NOT (NULL (LEFT  SUBTREE)))
                              (NOT (NULL (RIGHT SUBTREE))))
                         ; The found node has both subtrees
                           (SETQ NODE
                             (RIGHTLIST (LEFT SUBTREE)))
                           (RPLACA SUBTREE UZEL)
                           (COND ( (NULL (RIGHT
                                            (LEFT SUBTREE)))
                                   (REPLACE (LEFT SUBTREE)
                                     (CADR (LEFT SUBTREE))
                                            TREE) )
                                 ( (NULL (LEFT
                                            (LEFT SUBTREE)))
                                   (REPLACE (LEFT SUBTREE)
                                     (CADDR (LEFT SUBTREE))
                                            TREE) )
                                 ( T (REPLACE (SHEET NODE
                                                 NIL NIL)
                                                NIL TREE) )
                           )
                       )
                       ( (NULL (RIGHT SUBTREE))
                         ; The found node has only a left subtree
                            (REPLACE SUBTREE (CADR SUBTREE)
                                     TREE)
                       )
                       ( (NULL (LEFT SUBTREE))
                         ; The found node has only the right subtree
                            (REPLACE SUBTREE (CADDR SUBTREE)
                                     TREE)
                       )
                 )
            )
      )
   ))
   ; --------------------------------
   ( DEFUN DELETE1 (LAMBDA ( ATM TREE ) )
   ; Removing an ATM node from a TREE
   ; (recursive deletion option)
      (COND ( (NULL TREE) NIL )
            ( (< ATM (ROOT TREE))
                 (LIST (CAR TREE)
                       (DELETE1 ATM (LEFT TREE))
                       (RIGHT TREE))
            )
            ( (> ATM (ROOT TREE))
                 (LIST (CAR TREE)
                       (LEFT TREE)
                       (DELETE1 ATM (RIGHT TREE)))
            )
            (  T  (COND ( (NULL (RIGHT TREE)) (LEFT  TREE) )
                        ( (NULL (LEFT  TREE)) (RIGHT TREE) )
                        (  T  (LIST (UD (LEFT TREE))
                                    (DELETE1
                                        (UD (LEFT TREE))
                                        (LEFT TREE))
                                    (RIGHT TREE)) )) )
      )
   ))
   ; ----------------------
   ( DEFUN UD (LAMBDA ( TREE ) )
   ; Helper function for the DELETE1 function
      (COND ( (NULL (RIGHT TREE)) (CAR TREE) )
            (  T  (UD (RIGHT TREE)) )
      )
   ))
   ; -----------------------
   (DEFUN TOP (LAMBDA (TREE)
   ; The TOP function returns the number of levels in the tree TREE
   ; (the root of the tree is at level zero)
      (COND ( (NULL TREE) -1 )
            (  T  (+ 1 (MAX (TOP (LEFT  TREE))
                            (TOP (RIGHT TREE)))) )
      )
   ))
   ; ----------------------
   (DEFUN MAX (LAMBDA (M N))
   ; The MAX function returns the larger of the numbers M and N
      (COND ( (> M N) M )
            ( T  N )
      )
   ))
   ; -------------------------
   (DEFUN NLIST(LAMBDA (TREE))
   ; The NLIST function returns the number of leaves in a TREE
      (COND ( (NULL TREE) 0 )
            ( (EQUAL (CDR TREE) (LIST NIL NIL)) 1 )
            (  T  (+ (NLIST (LEFT TREE)) (NLIST (RIGHT TREE))) )
      )
   ))
   ; -------------------------
   ( DEFUN TCOPY (LAMBDA ( TREE ) )
   ; The TCOPY function returns a copy of the TREE
      (COND ( (ATOM TREE) TREE )
            (  T  (CONS (TCOPY (CAR TREE))
                        (TCOPY (CDR TREE))) )
      )
   ))
   ; -----------------------------
   (DEFUN PRETREE (LAMBDA (A TREE)
   ; The PRETREE function extracts all nodes preceding a given element A
   ; from the tree TREE into a separate tree
      (COND ( (NULL TREE) NIL )
            ( (< (CAR TREE) A)
                    (LIST (CAR TREE) (CADR TREE)
                          (PRETREE A (CADDR TREE))) )
            (   T   (PRETREE A (CADR TREE)) )
```

```
    )
))
; ------------------------------
( DEFUN POSTTREE (LAMBDA ( A TREE )
; The POSTTREE function extracts into a separate tree from the tree
; TREE all nodes following a given element A
   (COND ( (NULL TREE) NIL )
         ( (< (CAR TREE) A)
                (POSTTREE A (CADDR TREE)) )
         (   T   (LIST (CAR TREE)
                       (POSTTREE A (CADR TREE))
                       (CADDR TREE)) )
   )
))
; ------------------------------------
(DEFUN UNITETREE (LAMBDA (TREE1 TREE2)
; The UNITETREE function combines two search trees
; TREE1 and TREE2 into one search tree
   (COND ( (NULL TREE1) TREE2 )
         ( (NULL TREE2) TREE1 )
         (   T   (LIST (CAR TREE1)
                       (UNITREE (PRETREE (CAR TREE1)
                                           TREE2)
                                (CADR  TREE1))
                       (UNITETREE (POSTTREE (CAR TREE1)
                                           TREE2)
                                (CADDR TREE1))) ) )
))
; ----------------------------
( DEFUN UNTREE (LAMBDA ( M TREE ) )
; "Dirty" breadth-first traversal of the TREE tree, starting
; from level 0 to level M
   (COND ( (EQ M 0) (CAR TREE) )
         (  T  (LIST (UNTREE (- M 1) TREE)
                     (SEE M TREE)) )
   )
))
; ---------------------------
(DEFUN UNTREE1(LAMBDA (TREE))
; Left-hand traversal of a tree TREE
    (REMBER NIL (LISTATOMS TREE))
))
; -------------------------
(DEFUN SEE (LAMBDA (N TREE))
; Traverses a TREE in breadth and creates a "dirty"
; list containing the N-th level nodes of the tree
   (COND ( (EQ N 0) (CAR TREE) )
         ( (EQ N 1)
              (LIST (CAR (CADR TREE)) (CAR (CADDR TREE)))
         )
         (  T  (LIST (SEE (- N 1) (CADR  TREE))
                     (SEE (- N 1) (CADDR TREE))) )
   )
))
; ----------------------------
(DEFUN LISTATOMS (LAMBDA (TREE)
; The LISTATOMS function returns a list of
; elements (including NIL !) contained in the search tree TREE
   (COND ( (NULL TREE) NIL)
         ( (ATOM (CAR TREE))
              (CONS (CAR TREE) (LISTATOMS (CDR TREE))) )
         (  T  (APPEND (LISTATOMS (CAR TREE))
                       (LISTATOMS (CDR TREE))) )
   )
))
```

```
; -----------------------------
(DEFUN REMEMBER (LAMBDA) (ATM LST)
; The REMBER function returns a list with all occurrences
; of the ATM element in the list LST removed
   (COND ( (NULL LST) NIL )
         ( (EQ ATM (CAR LST)) (REMBER ATM (CDR LST)) )
         (  T  (CONS (CAR LST) (REMBER ATM (CDR LST))) )
   )
))
; -------------------------------
(DEFUN APPEND (LAMBDA (LST1 LST2)
; The APPEND function returns a list consisting of
; the elements of LST1 appended to LST2
   (COND ( (NULL LST1) LST2 )
         ( (NULL LST2) LST1 )
         (  T  (CONS (CAR LST1)
                     (APPEND (CDR LST1) LST2)) )
   )
))
```

Tasks for independent solution

When working with binary trees, you need to be ***able to***:

- ***build*** a binary search tree;
- ***traverse*** the constructed tree (three methods);
- ***find*** a specified node in a tree;
- ***add*** a vertex to a tree;
- ***remove*** a vertex from a tree.

Construction

1. Describe a function **(Copy T T1)** that constructs a tree **T1 -** a copy of the tree **T** .
2. Describe a function that, given a tree **T,** constructs a binary search tree consisting of the leaves of the tree **T**.
3. Construct a binary search tree for a given set of integers and number its vertices according to their traversal in internal order.
4. Construct a binary search tree for a given set of integers and number its vertices in accordance with the order of the direct traversal of this tree.
5. Construct a binary search tree for a given set of integers and number its vertices according to their order during the backward traversal of this tree.
6. (*) The text file **PROG** contains a program in **Pascal** (without syntax errors). It is known that in this program each identifier (service word or name) contains no more than 9 Latin letters and/or digits. Print in alphabetical order all the different identifiers of this program, indicating for each of them the number of its occurrences in the program text. To store identifiers, use a binary search tree whose elements are pairs - identifier and the number of its occurrences in the program text.

Modification

1. Write a function that adds a new node with element **E to the tree T** if it was not in **T** .
2. Write a function that replaces the values of all negative elements of the vertices in the tree **T** with their absolute values (the information field of the tree vertex contains real numbers).
3. Write a function that removes all nodes with the same elements from a non-empty tree **T** (use a set data structure)
4. Write a function that removes from a non-empty tree **T** vertices with maximum and minimum elements (the information field of a tree vertex contains real numbers).
5. Write a function that removes from a non-empty tree **T** all vertices with positive elements (the information field of a tree vertex contains real numbers).

Predicate

1. Write a function that determines whether a node containing an information field **E** appears in a tree **T** twice.
2. Write a function that checks whether a node containing an information field **E** is in the right or left subtree of a tree **T**.
3. Write a function that checks whether an element from the leftmost leaf of a non-empty tree **T** matches an element from the rightmost leaf of the same tree.
4. Determine whether it is possible to get from one top of a tree to another by moving along the branches toward the leaves.
5. Check whether a given binary search tree is an AVL tree.
6. Two trees are called ***isomorphic*** if one can be mapped into the other by changing the order of the children of its nodes. Recognize the isomorphism of two given trees $\mathbf{T_1}$ and $\mathbf{T_2}$.

Count

1. Describe a function that determines the number of occurrences of a vertex with a given element **E** in a tree **T** .
2. Describe a function that calculates the sum of the elements of all the vertices of a non-empty tree **T** (the information field of a tree vertex is of type **Real**).
3. Describe a function that determines the maximum depth of a non-empty tree **T** , i.e. the number of branches in the longest path from the root of the tree to the leaves.
4. Describe a function that counts the number of vertices at **the N-th** level of a non-empty tree **T** (the root is considered to be the vertex of level 0).
5. Write a function that determines the number of occurrences of an element **E** in the information fields of the vertices of the left subtree of a tree **T.**
6. Write a function that calculates the arithmetic mean of all elements of a non-empty tree **T** (the information field of the tree top contains real numbers).
7. Write a function that finds in a non-empty tree **T** the length (number of branches) of the path from the root to the nearest vertex with element **E** .
8. Determine the height of a tree **T** using the function that determines the path from the root to a given node.
9. Determine the total path of a given tree **T** using the path function from the root to a given node.
10. Write a function that selects all different English letters from a tree **T** (the information field at the top of the tree contains the character).
11. Write a function that determines the maximum element of all leaves of a non-empty tree **T** (the information field of the tree top contains integers).
12. Write a function that outputs the contents of the information field of all ***internal*** tree nodes.
13. Write out all vertices located at the same level **K** using the function for determining the path from a given vertex to the root.
14. Represent the set of integers as a binary search tree and, based on this representation, sort this set in descending order.

In the next step we will present ***problems on using imperative programming style***.

Step 118.
Practical lesson #8. Imperative programming. Integer arithmetic

In this step we will look at ***problems that involve the imperative programming style and integer arithmetic***.

Fragment of theory

1. Assigning values

The SET function binds the value of the argument **X** to the calculated value **Y**. The syntax of the function is:

```
   (SET X Y)
```

where **X** is an atom, **Y** is an **S** expression.

The SET function can be used to bind the value of an **S** -expression **Y** to an atom **X.** Note that **(SET X Y)** is equivalent to the assignment operator **X:=Y** in imperative programming languages.

Note that the **SET** function evaluates both arguments, meaning that it can be used to associate a **Y** value with a name that is also obtained by evaluation. For example:

```
  $ (SET (CAR (A B C)) 25)
  25
  $ A
  25
```

You can also associate a symbol with its value using the **SETQ** function. This function differs from the **SET** function in that it only evaluates its second argument. The letter **Q (Quote)** in the function name reminds you of the automatic blocking of the first argument.

Thus, **(SETQ A B)** is equivalent to **(SET (QUOTE A) B)**.

2. Organization of cycles

A representative of the group ***of iteration functions*** in the **LISP** language is the **LOOP** function, which in general has the form:

```
  (LOOP
      Expr1
      Expr2
       ...
      ExprN
  )
```

where any S -expressions or special constructions of the form **(CE $_1$ E $_2$ ... E $_k$)** can be used as arguments , where **C, E $_1$, E $_2$, ..., E $_k$** are **S** -expressions.

In the latter case, the list is used as the point of analysis ***of the loop termination condition***. If the evaluation of **the S** -expression **C** returns the value **NIL** , the iteration process continues, otherwise it is terminated, but first **the S** -expressions **E $_1$, E $_2$, ..., E $_k$** are evaluated sequentially and the last of the obtained values is returned as the value of the **LOOP** function.

Demonstration examples

Example 1. Finding the sum of the elements of a numerical list.

```
  ; ------------------------------------------- ;
  ; An example of imperative programming style ;
  ; ------------------------------------------- ;
  (DEFUN SUM1(LAMBDA (LSTS))
     (SETQ S 0)
     (LOOP
        ; Exit the loop when the list is empty ;
        ( (NULL LST) S )
        (SETQ S (+ S (CAR LST)))
        (SETQ LST (CDR LST))
     )
  ))
```

```
; ------------------------------------------- ;
; An example of functional programming style ;
; ------------------------------------------- ;
(I AM DEAD (LAMBDA (LST)
   (COND ( (NULL LST)  0 )
         (  T   (+ (CAR LST)  (SUM  (CDR LST))) )
   )
))
```

Example 2. Finding the sum of squares of integers from 1 to **N.**

```
; ----------------------------------------- ;
; An example of imperative programming style ;
; ----------------------------------------- ;
(DEFUN SUM (LAMBDA (N))
   (SETQ S 0)
   (SETQ I 1)
   (PRIN1 "The sum of the squares of numbers from 1 to N is: ")
   ( LOOP
       ; Exit from the loop when N=0 ;
       ( (EQ N 0)  0 )
       ; Exit from the loop when I>N ;
       ( (> I N)  S )
       (SETQ S (+ S (* I I)))
       (SETQ I (+ I 1))
   )
))
```

Example 3. The **FACTORIAL** function returns the factorial of the number **NUM** in the **ANS** variable .

```
; ----------------------------------------- ;
; An example of imperative programming style ;
; ----------------------------------------- ;
(DEFUN FACTORIAL (LAMBDA) (NUM ANSWER)
   (SETQ ANS 1)
   (LOOP
      ( (EQ NUM 0) ANS )    ; Exit from loop when NUM=0 ;
      (SETQ YEARS (* NUM YEARS))
      (SETQ NUM (- NUM 1))
   )
))
; ------------------------------------------- ;
; An example of functional programming style ;
; ------------------------------------------- ;
(DEFUN FACT (LAMBDA (X)
   (COND ( (ZEROP X)  1 )
         (  T   (* X (FACT (- X 1))) )
   )
))
```

Example 4. The **POWER** function returns **NUM1** raised to the power of **NUM2** . **NUM3** is the result of the **POWER** function.

```
; ----------------------------------------- ;
; An example of imperative programming style ;
; ----------------------------------------- ;
(DEFUN POWER (LAMBDA) (NUM1 NUM2)
   ( (AND (ZEROP NUM1) (MINUSP NUM2))
         (PRINT "Result does not exist!")
   (SETQ NUM3 1)
```

```
    (LOOP
       (SETQ NUM2 (DIVIDE NUM2 2))
       ( ( (EQ (CDR NUM2) 1)
                      (SETQ NUM3 (* NUM1 NUM3)) ) )
       ( SETQ NUM2 (CAR NUM2) )
       ( (ZEROP NUM2) NUM3 )
       ( SETQ NUM1 (* NUM1 NUM1))
    )
 ))
 ; --------------------------------------------- ;
 ; An example of functional programming style ;
 ; --------------------------------------------- ;
 (DEFUN EXPT (LAMBDA (NUM1 NUM2)
    (COND ( (ZEROP NUM2) 1 )
          ( (ZEROP (- NUM2 1)) NUM1 )
          (  T   (* NUM1
                       (EXPT NUM1 (- NUM2 1))) )
    )
 ))
```

Example 5. The **NTH** function returns a list obtained by removing the first **NUM** elements from the list **LST**.

```
 ; ------------------------------------------- ;
 ; An example of imperative programming style ;
 ; ------------------------------------------- ;
 ( DEFUN NTH (LAMBDA ( LST NUM )
    (LOOP
       (SETQ LST (CDR LST))
       (SETQ NUM (- NUM 1))
       ( (ZEROP NUM) LST )    ; Exit from loop when NUM=0 ;
    )
 ))
 ; --------------------------------------------- ;
 ; An example of functional programming style ;
 ; --------------------------------------------- ;
 (DEFUN NTH_1(LAMBDA (LST NUM))
    (COND ( (ZEROP NUM) LST )
          (  T   (NTH_1 (CDR LST) (- NUM 1)) )
    )
 ))
```

Example 6. The **GCD** function returns the greatest common divisor of numbers **NUM1** and **NUM2**.

```
 ; ------------------------------------------- ;
 ; An example of imperative programming style ;
 ; ------------------------------------------- ;
 (DEFUN GCD (LAMBDA (NUM1 NUM2 NUM3)
    (LOOP
       ( (ZEROP NUM2) NUM1 )
       (SETQ NUM3 NUM2)
       (SETQ NUM2 (MOD NUM1 NUM2))
       (SETQ NUM1 NUM3)
    )
 ))
 ; --------------------------------------------- ;
 ; An example of functional programming style ;
 ; --------------------------------------------- ;
 (DEFUN GCD_1(LAMBDA (M N))
 ; The function represents a formal notation of the algorithm
 described by Euclid (Elements, book
 7, theorem 2) ;
    (COND ( (EQ M N) M )
          ( (< M N) (GCD_1 N M) )
          (  T  (GCD_1 (- M N) N) )
     )
  ))
```

Example 7. A predicate that allows one to determine whether a given integer **N** is prime.

```
  ; ------------------------------------------- ;
  ; An example of imperative programming style ;
  ; ------------------------------------------- ;
  (DEFUN SIMPLE (LAMBDA (N)
     (SETQ FLAG 1)
     (COND ( (AND (EQ (MOD N 2) 0) (NOT (EQ N 2))) PRIN1 "Not simple!" )
           (  T  ( (SETQ I 3)
                  (LOOP
                     ( (< (CAR (DIVIDE  N 2) ) I) )
                     ( (EQ (MOD N I) 0)
                            (SETQ FLAG 0) )
                     ( SETQ I (+ I 2) )
                  )
                  (COND ( (EQ FLAG 0) PRIN1 "Not simple!" )
                        ( T PRIN1 "Simple!" ))
                  )
           )
     )
  ))
  ; --------------------------------------------- ;
  ; An example of functional programming style ;
  ; --------------------------------------------- ;
  (DEFUN ISPRIM (LAMBDA (N))
     (COND ( (EQ N 1) NIL )
           ( T (ISPR N 2) )
     )
  ))
  ; --------------------- ;
  (DEFUN ISPR (LAMBDA)
     (COND ( (EQ M N)          T  )
           ( (EQ (MOD N M) 0) NIL )
           ( T (ISPR N (+ M 1)) )
     )
  ))
```

Example 8. Some more useful functions...

```
  (DEFUN STRING-NUM (LAMBDA (X)
  ; Convert a list of digits to a list containing ;
  ; the corresponding single-digit numbers ;
     (COND ( (NULL X) NIL )
           (  T  (CONS (POSITION (CAR X)
                                 (UNPACK 123456789))
                       (STRING-NUM (CDR X))
                 )
           )
     )
  ))
  ; ---------------------------- ;
  (DEFUN POSITION (LAMBDA (X LST)
  ; The POSITION function returns the position of atom X in ;
  ; a single-level list LST (the first element has ;
  ; index 1). If the element is not in the list, the function ;
  ; returns 0 ;
     (COND ( (NULL LST)       0 )
          ( (EQ X (CAR LST)) 1 )
          ( (MEMBER X LST)
                (+ 1 (POSITION X (CDR LST))) )
          (  T  0 )
      )
   ))
   ; ----------------------- ;
   (DEFUN NUMBER (LAMBDA (LST))
   ; Given a numeric list LST containing single-digit ;
   ; numbers. "Build" an integer from the elements of the given ;
   ; list ;
      (COND ( (NULL LST) 0 )
            (  T  (+ (* (CAR LST)
                             (STAGE 10
                                     (-
                                       (LENGTH LST) 1))
                        )
                        (NUMBER (CDR LST))
                  )
            )
      )
   ))
   ; ----------------------- ;
   ( DEFUN STEP (LAMBDA ( X A )
   ; Raising an integer X to a non-negative integer ;
   ; power A ;
      (COND ( (ZEROP A) 1 )
            ( (ZEROP (- A 1)) X )
            (  T  (* (STEPEN X (- A 1)) X) )
      )
   ))
```

Tasks for independent solution

1. Write a program that, when executed, determines whether the digit 2 is included in the integer **n** .
2. Find a triple of consecutive natural numbers such that the sum of their squares is equal to the sum of the squares of the two following natural numbers. You have probably seen the painting by N.P. Bogdanov-Belsky (1868-1945) "Mental arithmetic". It depicts a lesson in oral problem solving in a school in the village of Tatevo, Smolensk region, a school founded and taught by Sergei Aleksandrovich Rachinsky since the 1970s. The painting depicts S.A. Rachinsky. Pay attention to the problem written on the blackboard: $\mathbf{(10^2 +11^2 +12^2 +13^2 +14^2)/365}$. Calculation shows that $\mathbf{10^2 +11^2 +12^2 = 13^2 +14^2 = 365}$.
3. ***A reversed number*** is a number written with the same digits but in reverse order. For example, 3805, reversed is 5083. ***A palindromic number*** is a number equal to its reversed number. For example, 121.5995 are palindromic numbers. Write a program to find several palindromic numbers less than 10001.
4. Write a program that returns the value **N** if **N** is a prime number, and "does nothing" otherwise (**N** is odd). A number is called ***prime*** if it has no divisors other than 1 and itself.
5. Is there a set of four consecutive natural numbers such that the sum of their squares is equal to the sum of the squares of the next three natural numbers? (Answer: $\mathbf{21^2 +22^2 +23^2 +24^2 =25^2 +26^2 +27^2}$)
6. Check whether a given natural number is divisible by 19 using the following divisibility test: a number is divisible by 19 if and only if the number of its tens added to twice the number of units is divisible by 19.
7. In one mathematical book, the typesetter typed 2592 instead of the numbers $2^5 * 9^2$. The proofreader missed this serious error, and the book was published in this form. Many mathematicians studied this book, all previous and subsequent calculations were checked, but no one noticed this obvious error, missed by the typesetter and proofreader. How could this happen? The fact is that $\mathbf{2^5 * 9^2 = 2592}$. This is apparently a unique numerical curiosity! Check it out.

Another solution was suggested by ***Alexey Anatolyevich Dorofeev*** from the city of Rostov-on-Don, a student of the Mechanics and Mathematics Department, specializing in "Applied Mathematics" at Rostov State University. Here is his letter:

"I accidentally went to http://it.kgsu.ru/Lisp/lisp0118.html; when I was reading "Problems for independent solution", I immediately noticed problem #7....

Actually, it was because of this problem that I got here - I typed “numerical curiosities” in the search (I am interested in mathematics and all sorts of curiosities) and followed the link. A very interesting problem, I must say (most importantly - beautiful; before this I thought that “there are no beautiful curiosities” - now I am convinced of the opposite). So, let's get back to the problem: "This is, apparently, the only one of its kind numerical curiosity! Check it out. “I thought about it all evening today, at least 4 hours. So, my answer: no, it is not the only one. And here is what I found: **35721 = (3^5) * 7 * 21** . Well, isn't it a curiosity? In my opinion, it is more curious than curious. Honestly, I have not read or looked at this anywhere, but I found it about half an hour ago after at least 4 hours of deep thought :)”

Check the truth of the solution proposed ***by Alexey Anatolyevich*** .

8. Find the number of “lucky” tickets with numbers from 000000 to 999999 inclusive. A ticket is considered “lucky” if the sum of the left three digits of the number is equal to the sum of the right three digits. Solve the problem by enumerating all possible ticket numbers.
9. Write a program for “processing” a given integer **n >= 10** into an integer whose notation differs from the notation of the number **n** by the permutation of the first and last digits.
10. What digit does the number 777^{777} end with? (Answer: 5)
11. Determine what digit the number $(...((7^7)^7)^7 ...)^7$ ends with, if it is known that raising to a power is repeated 77 times.
12. Prove that $999993^{1999} - 777777^{1997}$ is a multiple of 5.
13. Is the number $7^{77} +1$ divisible by 5?
14. Find the last digit of the following numbers: 6^{1971}, 9^{1971}, 3^{1971}, 2^{1971}.
15. Prove that the difference $9^{512} - 7^{512}$ is divisible by 10.
16. How many integers are there less than 1000 that are not divisible by either 5 or 7? How many of these numbers are not divisible by either 3, 5 or 7? (Answer: 686; 457)
17. Check the validity of the following statement using several examples: if the number **p** is prime and **p** is greater than 100 but less than 200, then the number **210-p** is also a prime number.
18. Is there a natural number **n** such that the number $\mathbf{2^n +15}$ is composite? (Answer: **n=7**)
19. The number 29 can be written as a sum of different powers of 2: $2^4 +2^3 +2^2 +1$. Write the given natural number as a sum of different powers of 2.
20. What digit does the decimal notation of the number $\mathbf{142+142^2 +142^3 +...+142^{20}}$ end with ?
21. It is known that the numbers $\mathbf{n^4 + n^3 + n^2 + n}$ are not divisible by 10. What digits can the decimal notation of the number **n** end with ? (Answer: 1 or 6)
22. The German mathematician ***M. Stiefel*** (1487-1567) claimed that numbers of the form $\mathbf{2^{2n+1} - 1}$, **n** belongs to **N** , are prime. Is he right?
23. What is the smallest natural number **m** that is divisible by 7 by numbers of the form: 1) $m^3 + 3^m$; 2) $m^2 + 3^m$.
24. Determine how many numbers from 1 to 1000 are divisible by 4 but do not have the digit 4 in their entry? (Answer: 162)
25. Find a two-digit number with the property that the cube of the sum of its digits is equal to the square of the number itself. (Answer: 27)
26. In which two-digit numbers is the sum of the digits doubled equal to their product? (Answer: 63,44,36)
27. Find a two-digit number equal to three times the product of its digits. (Answer: 15.24)
28. Integer numbers from 1 to 100 are written out in a row. How many times do the numbers appear in this entry: a) zero, b) one, c) three.
29. Write the smallest three-digit number multiple of 3 so that its first digit is 6 and all digits are different.
30. Find all three-digit numbers that are equal to the sum of the factorials of their digits.
31. Add three digits to 523... so that the resulting six-digit number is divisible by 7, 8 and 9.
32. Determine whether a given integer **p** is prime using ***Wilson's*** theorem (1770): An integer **p** is prime if and only if **(p - 1)! + 1** is divisible by **p** . Note that this theorem was proved ***by Lagrange*** in 1773.
33. Indicate such **n** that the number $\mathbf{n^4 +(n+1)^4}$ is composite. (Answer: 5)

34. Let **M(a,b,c,...,k)** denote the least common multiple, and **D(a,b,c,...,k)** denote the greatest common divisor of numbers **a,b,c,...,k** . Write a program that calculates **M(a,b,c)** using the formula: **M(a,b,c) = a*b*c*D(a,b,c)/(D(a,b)*D(b,c)*D(a,c))**
35. Natural numbers **m** and **n** are relatively prime and **n<m** . Which number is greater: **[1*m/n]+[2*m/n]+...+[n*m/n]** or **[1*n/m]+[2*n/m]+...+[m*n/m]** and by how much? (Answer: the first sum is greater by **mn**)
36. For each natural number **k<=10,** find the smallest natural number **n** for which $k*2^{2^n}+1$ is a composite number.

```
           k:  1 2 3 4 5 6 7 8 9 10
Answer:
           n:  5 1 2 2 1 1 3 1 2  2
```

37. Write a program to find all natural numbers less than **n** , whose square sum of digits is equal to **m** .
38. Can a given integer **p** be represented as the sum of two squares of integers?
39. Write a program to find, given an integer **n>7,** a pair of non-negative integers **a** and **b** such that **n=3a+5b** .
40. ***Goldbach*** suggested that every even number greater than or equal to 4 can be represented as the sum of two primes. This suggestion has not yet been proven or disproved. Write a program to test this hypothesis for a given even number. The result of executing the program should be the output of the number itself if a pair of prime terms could not be found, and the output of a pair of corresponding primes if such a pair was found.
41. Create a program to search among the numbers **n, n+1, ..., 2n** for so-called ***twins*** , i.e. two prime numbers whose difference is 2.
42. Find at least one solution to the equation $a^2 + b^4 = c^5$ in positive integers. (Answer: 256, 64, 32)
43. Is it true that for any natural **n** the number $n^3 + 5n-1$ is prime? (Answer: no, **n=6**)
44. Find the arithmetic mean of all divisors of a given number **n** (including 1 and **n**). Check that it lies between the square root of **n** and **(n+1)/2**.
45. Find the number of all divisors of a given natural number **n**. Check that the product of all divisors is equal to $n^{s/2}$, where **s** is the number of all divisors.
46. A natural number is called ***palindromic*** if it reads the same on both sides, for example, 171. Take any number. If it is not a palindrome, then turn it over and add it to the original. If it does not turn out to be a palindrome, then do the same with it, and so on, until it turns out to be a palindrome. For all numbers from 1 to 1000, find how many steps it takes to get a palindrome and which one.
47. Calculate how many zeros the product of all numbers from 1 to 100 inclusive ends with. (Answer: 24)
48. A natural number is called ***perfect*** if it is equal to the sum of all its divisors, not counting itself (for example, 6=1+2+3 is a perfect number). Write a program that checks whether a given number is perfect.
49. Write a program that finds perfect numbers.
    Apply the following theorem.
    Euclid's theorem (Elements, Book 9).
    When the number $p=2^{n+1} - 1$ is prime, $p*2^n$ is perfect.
    Nicomachus of Gerasa indicated ***the first three perfect numbers***: 6, 28, 496.
50. ***Amicable numbers*** are a pair of numbers that have the following property: the sum of the proper divisors of the first is equal to the second number, and the sum of the proper divisors of the second number is equal to the first number.
    For example, the sum of the divisors of 220 is 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284, and the sum of the divisors of 220 and 284 is an amicable pair.
    The second amicable pair (1184 and 1210) was found in 1867 by the sixteen-year-old Italian ***B. Paganini*** .
    Write a program to find amicable numbers using the method indicated in the 9th century by the Arab mathematician ***Sabit ibn Qurra*** .
    Sabit's theorem . If all three numbers $p = 3*2^{n-1} - 1$, $q = 3*2^n - 1$, $r = 9*2^{2n-1} - 1$ are prime, then the numbers $A = 2^n*p*q$ and $B = 2^n*r$ are amicable.
    For **n=2,** the numbers **p=5, q=11** and **p=71** are prime, and we get the pair of numbers 220 and 284, discovered ***by Pythagoras*** . However, Sabit's theorem gives amicable numbers for other **n** as well , for example:

```
              n=4                n=7
              A=17296 A=9363584
              B=18416 B=9437056
```

It is now known that these three cases exhaust all the values of **n<=20000** for which the indicated method yields amicable numbers.

51. Solutions to the equation **$a^2 + b^2 = c^2$** in positive integers are called ***Pythagorean triples*** . For example: (3,4,5), (5,12,13), (41,140,149).
    We will look for the simplest Pythagorean triples **(a,b,c)**, that is, those for which l.o.d.(a,b,c)=1. The solution to this problem was indicated by ***Diophantus of Alexandria*** (c. 250 AD): if **n** and **m** are two relatively prime integers (positive) numbers whose difference **nm** is positive and odd, then **($2*n*m$, $n^2 - m^2$, $n^2 + m^2$)** is the simplest Pythagorean triple, and any of these triples can be found in this way.
    Write a program to find several Pythagorean triples using Diophantus's theorem.
52. ***Fermat's problem***. Find a cube that, when added to all its proper divisors, is a square (for example, the sum of the number 343 and all its proper divisors 343+49+7+1=400 is a square).
53. ***Fermat's problem***. Find a square which, when summed with all its proper divisors, gives a cube.
54. Write a program that calculates the number of square-free numbers less than or equal to **X.** A number is called ***square-free*** if it is not divisible by any square of a prime number.
55. Find the values **n<=N** that have the following property: the fifth power of the sum of the digits of the decimal notation of the number **n** is equal to **n^2**. (Answer: 1 and 243)
56. What is the minimum **N** for a number of the form ($9^N - 7^N$) to be divisible by 10? (Answer: 512)
57. Write a program for compiling a table of prime numbers not exceeding a given integer **N** using ***the sieve of Eratosthenes*** .
    The method is as follows. Write out the numbers **1, 2, ..., N (1)** The first number in this series greater than one is 2. It is divisible only by 1 and itself and, therefore, it is prime.
    Cross out from the series (1) (as composite) all numbers multiples of 2, except for the number 2 itself. The first number following 2 that is not crossed out is 3. It is not divisible by two (otherwise it would be crossed out). Therefore, 3 is divisible only by 1 and itself and, therefore, it will also be prime. Cross out from the series (1) all numbers multiples of 3, except for 3 itself. The first number following 3 that is not crossed out is 5. It is divisible by neither 2 nor 3 (otherwise it would be crossed out). Therefore, 5 is divisible only by 1 and itself, and therefore it will also be prime. And so on.
    It is easy to prove that the table of prime numbers not exceeding **N** is complete as soon as all composite multiples of primes ***not exceeding the square root*** of **N** are crossed out .
58. Write a program for a computer game called "Bulls and Cows". The rules of the game are extremely simple. One of the partners is the leader. He thinks of any four-digit number. It is only required that none of the digits in this number be repeated. The task of the other partner (let's just call him the player) is to guess this number. The player names trial numbers. The game ends when the number conceived by the leader is named. The fewer moves-attempts the player needs to do this, the better he plays. The leader helps the player with hints. Let's consider what these hints are using an example.
    Let's assume that the number 9480 has been conceived. Let's call it ***the code*** . As prescribed by the rules of the game, all the digits in the code are different. The player names his trial number 7984. The leader compares it with the code 9480. Since the numbers do not match, the game is not over. The leader tells the player the number of "bulls" and "cows" contained in the trial number he has proposed. "Bulls" are those numbers that match the corresponding numbers of the intended code in value and position. In our example, there is one "bull" - the number 8. "Cows" are numbers that match the numbers of the intended code, but are in other positions. In the number 7984, there are two "cows": 9 and 4. It is clear that guessing the code or naming four "bulls" is the same thing.
59. Write a program to determine the corresponding day of the week based on known integers: **J** - day, **M** - month, **A** - year, using ***M. Lenoir's*** method , which is as follows:
    - calculate the value of **N** : if the month is January or February of a leap year, then **N=1** ; if the month is January or February of a regular year, then **N=2** ; in other cases N=0.
      To find out whether a year is a leap year, you can do the following:
        - break the year into two parts **A1** and **A2** : **A1** contains the 2 most significant digits of the year, **A2** contains the 2 least significant digits of the year;
        - if **A2=0** and **A1** is divisible by 4, then the year is a leap year;

        - if **A2** is not equal to 0, then the year is a leap year if **A2** is divisible by 4;
    - calculate the "code" of day **C** using the formula **C = integer (365.25 * A2) + integer (30.56 * M) + J + N** ;
    - calculate the remainder **S** from dividing **C** by 7: if **S=0** , then the day is Wednesday, if **S=1** , then the day is Thursday, if **S=2** , then the day is Friday, ..., if **S=0** , then the day is Tuesday.

60. Write a program to find prime numbers using ***Lehmer's*** theorem (1931): A number $2^p - 1$ (where $p>2$) is prime if and only if it divides v_{p-1} , where $v_1 = 4$ and $v_{n+1} = v_n^2 - 2$.
    As an illustration, note that $2^3 - 1$ divides $v_2 = 14$; $2^4 - 1$ does not divide $v_3 = 194$; $2^5 - 1$ divides $v_4 = 37634$.
61. Find a natural number from 1 to 10000 with the maximum sum of divisors.
62. ***Simon's*** factorial conjecture states that only four factorials are the product of three consecutive integers. Here are two of them: 4!=2*3*4, 5!=4*5*6. Write a program that would help you find the next natural number that has the specified property. Could you disprove Simon's conjecture?

Problems 63-81 are borrowed from [1].

63. Find all numbers consisting of three different digits, each of which is divisible by the square of the sum of its digits. (Answer: 162, 243, 324, 392, 45, 512, 65, 648, 810, 972)
64. Find all five-digit squares for which the sum of the numbers formed by the first two digits and the last two digits is equal to the exact cube. (Answer: $152^2 = 23104$; $23 + 04 = 27 = 9^3$; $237^2 = 56169$; $56 + 69 = 125 = 5^3$; $251^2 = 63001$; $63 + 01 = 64 = 4^3$)
65. What number is formed from five consecutive digits (not necessarily in order) such that the number formed by the first two digits, multiplied by the middle digit, gives the number formed by the last two digits? (consider the number 01234 as a five-digit number). (Answer: 3412 = 03412; 03*04=12; 4312 = 04312; 04*03=12; 13452; 13*04=52)
66. I have five cards with the numbers 1,3,5,7 and 9 on them. How can I arrange them in a row so that the product of the number formed by the first pair of cards and the number formed by the last pair of cards, minus the number on the middle card, equals the number made up of repetitions of the same digit (i.e. I get a number in which all the digits are the same).
67. Can you arrange the numbers 1,2,3,4,5,7,8,9 in two groups of four digits each so that the sums of the numbers made up of the digits in each group are equal to each other?
68. Can you arrange the numbers 1,2,3,4,5,6,7,8 in two groups of four digits each so that the sums of the numbers made up of the digits in each group are equal to each other? (Answer: 1,2,7,8 and 3,4,5,6)
69. Find all numbers whose cube root coincides with the sum of the digits. Prove that the digits of the cube root and the digits of the sum of the digits of a number coincide if the number has no more than six digits. (Answer: $1 = 1^3$, 1 = 1; $512 = 8^3$, 5+1+2 = 8; $4913 = 17^3$, 4+9+1+3 = 17; $5832 = 18^3$, 5+8+3+2 = 18; $17576 = 26^3$, 1+7+5+7+6 = 26; $19683 = 27^3$, 1+9+6+8+3 = 27)
70. Some natural numbers can be represented as a sum of cubes of non-negative integers: for example, $9=2^3+1^3$, $27=3^3+0^3$. Create an algorithm that finds the smallest natural number that has two different such representations. (The representations $9=2^3+1^3$ and $9=1^3+2^3$ are considered the same).
71. Find the smallest natural number greater than 2 that is simultaneously the sum of two squares of natural numbers and the sum of two cubes of natural numbers. (Answer: $65=8^2+1^2=4^3+1^3$)
72. Find the smallest natural number greater than 1 for which the sum of the squares of consecutive natural numbers from 1 to **n** would be a square of a natural number.
73. Find the smallest natural **number n** for which $n^4 + (n+1)^4$ is a composite number.
74. Find all numbers of the form $2^n - 1$ (where **n** is a natural number) that are not greater than a million and are products of two prime numbers.
75. Find the five smallest natural numbers **n** for which $n^2 + 1$ is the product of three distinct prime numbers, and find a natural number **n** for which $n^2 + 1$ is the product of three distinct prime numbers.
76. Find the five smallest natural numbers **n** for which the number $n^2 - 1$ is the product of three different prime numbers.
77. Find five prime numbers that are sums of biquadratics (fourth powers) of natural numbers.
78. Find the smallest natural number **n** such that **n** is divisible by $2^n - 2$, but not divisible by $3^n - 3$.
79. Find the smallest natural number **n** such that **n** is not divisible by $2^n - 2$, but **n** is divisible by $3^n - 3$.
80. Find the two smallest composite numbers **n** such that **n** is divisible by $2^n - 2$ and **n** is divisible by $3^n - 3$.

81. Find the five smallest natural numbers **n** for which each of the numbers **n, n+1, n+2** is the product of two different prime numbers. (Answer: 33=3*11, 34=2*17, 35=5*7; 85=5*17, 86=2*43, 87= 3*29; 93=3*31, 94=2*47, 95= 5*19; 141=3*47, 142=2*71, 143=11*13; 201=3*67, 202=2*101, 203= 7*29; 213=3*71, 214=2*107, 215= 5*43; 217=7*31, 218=2*109, 219= 3*76; 301=7*43, 302=2*151, 303= 3*101)

Problems 82-90 are taken from Klimenchenko D.V. Problems in Mathematics for the Inquisitive: Book for 5th-6th Grade Secondary School Students. - Moscow: Education, 1992. - 192 p.

82. How many two-digit numbers are there in which: a) there is at least one digit 3; b) the number of tens is less than the number of units? (Answer: a) 18; b) 36)
83. How many natural numbers, not exceeding 1000, are there such that each subsequent digit is greater than the previous one? (Answer: 120)
84. Find a four-digit number whose two middle digits form a number that is 5 times greater than the number of thousands and 3 times greater than the number of units. (Answer: 3155)
85. How many three-digit numbers can be written without using the number 7? (Answer: 648)
86. The distance between two cities in kilometers is expressed by a two-digit number such that its left digit is equal to the difference between this number and the number written with the same digits, but in reverse order. What is this distance? (Answer: 98 km)
87. All natural numbers from 1 to 100 inclusive are divided into 2 groups - even and odd. Determine in which of these groups the sum of all the digits used to write the numbers is greater and by how much. (Answer: The sum of all the digits of odd numbers is greater than the sum of all the digits of even numbers by 49)
88. The number of students studying in grades V-VI of a school is expressed by a three-digit number. From the digits of this number (without repeating them) it is possible to make 6 different two-digit numbers, the sum of which is twice the number of students in grades V-VI. How many students are there in these classes? (Answer: 198 students)
89. Find the smallest natural number that when multiplied by 2 produces a perfect square, and when multiplied by 3 produces a perfect cube. (Answer: 72)
90. Find the sum of three-digit numbers, each of which is the product of four unequal prime numbers. (Answer: 10494)
91. ***Automorphic*** numbers are those that are contained in the last digits of their square. For example: $5^2 = 25$, $25^2 = 625$. Write a program to find several automorphic numbers.
92. Find $\mathbf{GCD(u_m, u_n)}$, where $\mathbf{u_m}$ and $\mathbf{u_n}$ are Fibonacci numbers, using the formula $\mathbf{GCD(u_m, u_n) = u_{GCD(m,n)}}$.
93. Let a Fibonacci number $\mathbf{u_n}$ be divisible by some prime number **p**, and let none of the Fibonacci numbers less than $\mathbf{u_n}$ be divisible by **p**. In this case we will call the number **p** ***a proper divisor of*** $\mathbf{u_n}$. Show that the Fibonacci numbers 1, 8, and 144 have no proper divisors.
94. Find a Fibonacci number that has several proper divisors (for example, for $\mathbf{u_{19}}$ such divisors will be numbers 37 and 113, for $\mathbf{u_{27}}$ - numbers 53 and 109, etc.). It is not at all clear how many Fibonacci numbers with two or more proper divisors there are!
95. Calculate the number of a member of the Fibonacci sequence with a pre-assigned proper divisor. (No formula is known yet that allows one to directly calculate the numbers of members with a pre-assigned proper divisor **p !)**
96. Given a sequence of Fibonacci numbers defined by the relations $\mathbf{U_1 = 1, U_2 = 1, U_n = U_{n-1} + U_{n-2}}$ **, n>2**. Check whether $\mathbf{U_{5k}}$ is divisible by 5, **k** =1, 2, ...
97. For a given integer m, find among the first $\mathbf{m^2 -1}$ Fibonacci numbers at least one that is divisible by **m**.
98. Find some prime Fibonacci numbers (it is not yet known whether the number of all prime Fibonacci numbers is finite or infinite).
99. Determine the number of the Fibonacci number whose proper divisor is the given prime **p**. Note that no formula is yet known that allows one to directly calculate the numbers of terms with a given proper divisor **p**.

The following tasks are compiled according to the monograph [2] using the following terminology.

A number written as a repetition of some digit is called ***a repdigit***. A number written with one or more units is called ***a repunit*** . We will denote by $\mathbf{R_n}$ a repunit containing **n** units in its notation.

The length of the period of a number **n** is the number of digits in the period of the decimal representation of the fraction **1/n**.

D.R.Khaprekar called a number that is a multiple of its sum of digits equal to **d** as ***the Harshad number*** 0 (**H**-number) for **d** . For example, 247 is **the H**-number for 13.

N is said to be ***a non-zero*** **H*-number*** for **d** if **N** is **an H**-number for **d** and if no digit of **N** is zero.

Let **S(n)** be the sum of the digits of the number **n** , and **Sp(n)** be the sum of the digits of all factors in the decomposition of the number into prime factors.

If **n** is composite and **S(n)=Sp(n)**, then **n** is called ***the Smith number*** or simply ***"Smith"*** . Let us check, for example, that the number 22 is a Smith number: 22=2*11, **S(22)=4, Sp(22)=4**.

100. Verify that the period length of a prime number not equal to three is equal to the number of units in the smallest repunit divisible by that prime number.
101. Check that any repunit divisible by a repunit $\mathbf{R_m}$ has the form $\mathbf{R_{mn}}$.
102. Verify that the period length of the product of two prime numbers is the least common multiple of their period lengths.
103. Determine the length of the period of a given composite number.
104. Indian scientist ***E.V. Krishnamurti*** discovered that among 1226 prime numbers greater than 5 and less than 10,000, the ratio of primes with even period length to primes with odd period length is 2.01 to 1. Check this fact.
105. ***Kaprekar's number*** (1981) is an **n**-digit integer whose square is the number for which the sum of the number of its rightmost n digits and the number of the remaining leftmost digits is **k** . For example, $142857^2 = 20408122449$. The sum 20408 + 122449 = 142857 is obviously a Kaprekar number. Find another Kaprekar number.
106. **A number M** is said ***to have the Maidy property*** if the period length of **M** is an even number and if the sum of the two halves of the period of an irreducible fraction with denominator **M** is ***a repdigit*** consisting of nines.
Show that every prime with an even period length has the Maidy property, but composite numbers with an even period length do not always have it. For example, 1/21 = 0 (047619). The equality 047 + 619 = 666 shows that the number 21 does not have the Maidy property.
107. Calculate some simple repunits.
108. Can a repunit be a power of some natural number?
109. Are there repunits that are divided into repunits?
110. Find a pair of repunites whose product is equal to a 100-digit palindrome.
111. What is the smallest repunit divisible by the square of 11?
112. The product of which two repunites is the number 123455554321?
113. Find the first 10 Smith numbers.
114. Let **L(p)** be the length of the period of a prime number. Find a prime number **p** such that $\mathbf{L(p)=L(p^3)}$.
115. Verify the statement that if **p** is a prime number other than 3, then $\mathbf{R}_{\mathbf{p}}$ divides only those primes whose period length is **p**.
116. ***Kaprekar's problem***. Find the smallest **H** -number and the largest non-zero **H** -number for a given d.
117. Check that if $\mathbf{m=3^j}$, then $\mathbf{R}_{\mathbf{m}}$ is the largest non-zero **H** -number for **m**.
118. Professors ***S. Olticar*** and ***K. Wayland*** from the University of Puerto Rico showed that if **n>2** and the repunit $\mathbf{R}_{\mathbf{n}}$ is prime, then $\mathbf{3304*R_n}$ is “smith”. Check this fact on any example.

[1] Beida A.A., Korolev S.A. Game tasks and computers: Book for students. - Mn.: Nar.sveta, 1991. - 144 p.
[2] Yates S. Repunites and decimal periods. - M.: Mir, 1992. - 254 p.

In the next step we will present ***tasks that expand the capabilities of the educational*** **LISP** interpreter.

Step 119.
Practical lesson #9. Extending the educational LISP interpreter

In this step we will consider ***tasks that expand the capabilities of the educational*** **LISP** interpreter.

Test examples for the training interpreter

```
1. <-- ((NLAMBDA1 (X) (CDR1 X)) (1 (TIMES1 1 2) 4))
   --> ((TIMES1 1 2) 4)
   <-- ((NLAMBDA1 (X) (CAR1 X)) ((TIMES1 1 2) (PLUS1 1
   4)))
   --> (TIMES 1 2)
2. <-- (DEFUN1 SUMMA (LAMBDA1 (LST) (COND1 ((EQUAL1 LST
   NIL1) 0) (T1 (PLUS1 (CAR1 LST) (SUMMA (CDR1 LST))))))
   --> (LAMBDA1 (LST) (COND1 ((EQUAL1 LST NIL1) 0) (T1
   (PLUS1 (CAR1 LST) (SUMMA (CDR1 LST))))))
   <-- (SUMMA (QUOTE1 (1 2 3)))
   --> 6
3. <-- (DEFUN1 FACT (LAMBDA1 (X) (COND1 ((EQUAL1 X 1) 1)
   (T1 (TIMES1 X (FACT (DIFFERENCE1 X 1)))))))
   --> (LAMBDA1 (X) (COND1 ((EQUAL1 X 1) 1) (T1 (TIMES1
   X (FACT (DIFFERENCE1 X 1))))))
   <-- (FACT 5)
   --> 120
4. <-- (DEFUN1 TEST (LAMBDA1 (X Y) (CONS1 X Y)))
   --> (LAMBDA1 (X Y) (CONS1 X Y))
   <-- (TEST A B)
   --> (A . B)
5. <-- (DEFUN1 COPY (LAMBDA1 (E) (COND1 ((ATOM1 E) E) (T
   (CONS1 (COPY (CAR1 E)) (COPY (CDR1 E)))))))
   --> (LAMBDA1 (E) (COND1 ((ATOM1 E) E) (T (CONS1 (COPY
   (CAR1 E)) (COPY (CDR1 E))))))
   <-- (COPY (QUOTE1 (A B C)))
   --> (A B C)
6. <-- (DEFMACRO1 AAA (NLAMBDA1 (G) (CAR1 G)))
   --> (NLAMBDA1 (G) (CAR1 G))
   <-- (AAA ((TIMES1 2 3) 4 5))
   --> (TIMES1 2 3)
```

Further, you can further expand the capabilities of the constructed interpreter, using your own mechanism for defining functions.

For example, let's define the **NULL1** function through the system primitives:

```
<-- (DEFUN1 NULL1 (LAMBDA1 (L) (EQUAL1 L NIL1)))
--> (LAMBDA1 (L) (EQUAL1 L NIL1))
```

and let's use it right away:

```
7. <-- (DEFUN1 MEMBER1 (LAMBDA1 (AL LST) (COND1 ((NULL1
   LST) NIL1) ((EQUAL1 AL (CAR1 LST)) LST) (T1 (MEMBER1
   AL (CDR1 LST))))))
   -->  (LAMBDA1 (AL LST) (COND1 ((NULL1 LST) NIL1)
   ((EQUAL1 AL (CAR1 LST)) LST) (T1 (MEMBER1 AL (CDR1
      LST)))))
      <-- (MEMBER1 (QUOTE1 B) (QUOTE1 (A B C)))
      --> (B C)
   8. <-- (DEFUN1 OF (LAMBDA1 (AL) (COND1 ((NULL1 L) NIL1)
      ((EQUAL1 (CAR1 L) A) (CDR1 L)) (T1 (CONS1 (CAR1 L)
      (From A (CDR1 L)))))))
      --> (LAMBDA1 (A L) (COND1 ((NULL1 L) NIL1) ((EQUAL1
      (CAR1 L) A) (CDR1 L)) (T1 (CONS1 (CAR1 L) (DEL A (CDR1
      L)))))
      <-- (DEL 5 (QUOTE1 (3 4 5)))
      --> (3 4)
   9. <-- (DEFUN1 EVERY (LAMBDA1 (L) (COND1 ((NULL1 L) NIL1)
      ((NULL1 (CDR1 L)) L) (T1 (CONS1 (CAR1 L) (EVERY (CDR1
      (CDR1 L)))))))))
      --> (LAMBDA1 (L) (COND1 ((NULL1 L) NIL1) ((NULL1 (CDR1
      L)) L) (T1 (CONS1 (CAR1 L) (EVERY (CDR1 (CDR1 L)))))))
      <-- (EVERY (QUOTE1 (A P R O L D)))
      (A R L)
  10. <-- (DEFUN1 APPEND1 (LAMBDA1 (L1 L2) (COND1 ((NULL1 L1)
      L2) ( T1 (CONS1 (CAR1 L1) (APPEND1 (CDR1 L1) L2)))))))
      --> (LAMBDA1 (L1 L2) (COND1 ((NULL1 L1) L2) ( T1 (CONS1
      (CAR1 L1) (APPEND1 (CDR1 L1) L2))))))
      <-- (APPEND1 (QUOTE1 (A P R)) (QUOTE1 (O L D)))
      (A P R O L D)
```

Tasks for extending the interpreter

1. Extend the tutorial interpreter to include interpretation:
   - a function that returns a copy of the list;
   - function that returns the largest element of a single-level numeric list **LST**;
   - functions that allow you to "glue" two lists.

2. Consider extending the training interpreter to include:
   - function **CDRN** such that **(CDRN NL)** will be equivalent to the function **(CD...DR L)**, whose name contains **N** letters **D** ;
   - a function that returns a list constructed from the first **N** elements of a given list **LST**;
   - functions that return the length of a list.

3. Determine the version of the training interpreter that includes the interpretation:
   - a function that writes the elements of a list in reverse order;
   - function that returns **the N-th** element of the list **LST**;
   - function that removes **the N-th** element of the list **LST**.

4. Complete the interpreter so that it includes interpretation:
   - function that inserts element **X** into **the N-th** position of the list **LST**;
   - functions that sort the list **LST**;
   - **the DISTL** (distribution to the left) function, the operation of which we will consider using an example:

```
    (A (B1 B2 ... BN)) --> ((A B1) (A B2) ... (A BN))
```

5. Extend the tutorial interpreter to include interpretation:

- **DISTR** function (distribution to the right), the operation of which we will consider using an example:

```
   ((A1 A2 ... AN) B) --> ((A1 B) (A2 B) ... (AN B));
```

- **COPY** functions such that **(COPY X N)** returns a list of **N** copies of the integer **X** . For example, **(COPY 3 5)** returns the list **(3 3 3 3 3)** ;
- **the LAST** function , which returns the last element of a list.

6. Define an extension of the training interpreter for the following functions:
   - function **DEPTH** such that **(DEPTH L)** returns the maximum depth of sublists of list **L** . Thus, if **L** is **(1 4 (2 6 (3 7) 8))**, then **(DEPTH L)** returns 3;
   - **LIST** functions;
   - a function that returns a list constructed from the last **N** elements of a given list.

7. Rewrite the interpreter so that it includes interpretation:
   - the function **INSLIST** , depending on three arguments **N, U** and **V** , inserts into the list **U** , starting from the **N-th** element, the list **V**;
   - functions that remove duplicate entries of elements in a list, for example: **(A B D D A) --> (A B D)**;

8. Extend the tutorial interpreter to include:
   - logical operations **NOT, AND** and **OR**;
   - logical operations: implication and equivalence.
   - predicate **ALL** (universal quantifier) and predicate **EXIST** (existential quantifier) with syntax: **(ALL XE)** and **(EXIST XE)**.

When evaluating the predicate **ALL,** the expression **E** is evaluated twice, first assigning the variable **T the value T**, and then assigning it the value **NIL** . The **AND** operation is applied to the results. Similarly, the predicate **EXIST** also requires evaluating the expression **E** for two different values of the variable. The **OR** operation is then applied to the results. Such an interpreter is a computational form of ***propositional calculus***.

From the next step we will start to give ***problems with solutions***.

Step 120.
Solved tasks. Tasks on list processing

In this step we will provide ***solutions to list processing problems***.

The tasks are listed in order of increasing difficulty. Try to find your own solutions and compare them with the suggested answers.

Task 1.

```
  ; ---------------------------------------------------- ;
```

```
; A function that returns a list of 3 elements, ;
; namely: a square, a cube, and the fourth power of N ;
; -------------------------------------------------- ;
(DEFUN SP(LAMBDA (N))
   (LIST (* N N)
         (* (* N N) N)
         (* (* (* N N) N) N))
))
```

Task 2.

```
; -------------------------------------------- ;
; Define the operations of addition and multiplication over ;
; quaternions ;
; ------------------------------------------- ;
(DEFUN KVA (LAMBDA (L M))
; L and M are lists of the form (XYZK) ;
; Addition of quaternions ;
   (LIST (+ (CAR L) (CAR M))
         (+ (CADR L) (CADR M))
         (+ (CADDR L) (CADDR M))
         (+ (CAR (CDDDR L))(CAR (CDDDR M)))
   )
))
; -------------------- ;
(DEFUN TIME (LAMBDA (L M))
; Quaternion multiplication ;
   (SETQ A (CAR L))
   (SETQ B (CADR L))
   (SETQ C (CADDR L))
   (SETQ D (CAR (CDDDR L)))
   (SETQ A1 (CAR M))
   (SETQ B1 (CADR M))
   (SETQ C1 (CADDR M))
   (SETQ D1 (CAR (CDDDR M)))
   (LIST (- (- (* A A1) (* B B1))
         (+ (* C C1) (* D D1)))
         (+ (+ (* A B1)(* B A1))
         (- (* C D1)(* D C1)))
         (+ (+ (* A C1)(* C A1))
         (- (* D B1)(* B D1)))
         (+ (+ (* A D1)(* D A1))
         (- (* B C1)(* C B1))))
))
```

Task 3.

```
;-------------------------------------------------- ;
; Write a function that returns the last element of the ;
; list. If the list is empty, return NIL ;
;-------------------------------------------------- ;
(DEFUN POSL (LAMBDA (L))
   (COND ( (NULL L) NIL )
         (  T  (CAR (REVERSE L)) )
   )
))
```

Task 4.

```
; -------------------------------------------- ;
; Given a numeric list A, determine
whether the sum of its elements is ; ; an even number ;
;------------------------------------------ ;
(DEFUN OTV (LAMBDA (LST)
   (COND ( (EQ (MOD (SUM LST) 2) 0) T )
         (  T  NIL  )
   )
))
; -------------------- ;
(I AM DEAD (LAMBDA (LST)
; Finds the sum of the elements of a numeric list ;
   (COND ( (NULL LST) 0 )
         (  T  (+ (CAR LST) (SUM (CDR LST))) ))
))
```

Task 5.

```
; ------------------------------------------ ;
; Implement the following algorithm in LISP, ;
; written in APL ;
; X <- 1 2 3 ;
; +/X ;
; ------------------------------------------ ;
(DEFUN (LAMBDA (LST)
   (COND
      ( (NULL LST) 0 )
      (  T  (+ (CAR LST)
               (SUM (CDR LST))) )
   )
))
```

Task 6.

```
; ----------------------------------------- ;
; Find an element of a numeric list that is equal to the sum ;
; of the first and last elements of the same list ;
; ----------------------------------------- ;
( DEFUN MAIN (LAMBDA ( LST ) )
   (COND ( (NULL LST) NIL )
         (  T  (MEMBER (+ (CAR LST)
                       (CAR (REVERSE LST))) LST) )
   )
))
```

Task 7.

```
; ------------------------------------------------ ;
; Construct a list containing only negative ;
; elements of the given numeric list LST ;
; ------------------------------------------------ ;
(NEGATIVE DEFUN (LAMBDA (LST)
   (COND ( (NULL LST) NIL)
         ( (MINUSP (CAR LST))
               (CONS (CAR LST) (NEGATIV (CDR LST))) )
         ( T (NEGATIVE (CDR LST)) )
   )
))
```

Task 8.

```
; ---------------------------------------------------- ;
; Define a function that constructs an N-level nested ;
; list whose element at the deepest level ;
; is the number N ;
; ---------------------------------------------------- ;
( DEFUN LUC (LAMBDA ( NX ) )
   (COND ( (ZEROP N) X )
         (  T  (LIST (LUC (- N 1) X)) )
   )
))
```

Task 9.

```
; ------------------------------------------------------ ;
; Replace negative elements of a numeric list with 0 ;
; ------------------------------------------------------ ;
(DEFUN ZAM (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (MINUSP (CAR LST))
                     (CONS 0 (ZAM (CDR LST))) )
         (  T  (CONS (CAR LST) (ZAM (CDR LST))) )
   )
))
```

Task 10.

```
; ---------------------------------------- ;
; Write a function that "cuts off" a list, ; ;
if it consists of more than N elements ;
; ---------------------------------------- ;
(DEFUN FIRSTN (LAMBDA (LST N)
   (COND ( (EQ N 1) (LIST (CAR LST)) )
         ( (< (LENGTH LST) N) LST )
         (   T   (CONS (CAR LST)
                       (FIRSTN (CDR LST)
                               (- N 1))) )
   )
))
```

Task 11.

```
; -------------------------------------------------- ;
; Find out if there are two identical elements in a list ;
; -------------------------------------------------- ;
( DEFUN POISK (LAMBDA ( LST ) )
   (COND  ( (NULL LST) NIL )
          ( (MEMBER (CAR LST) (CDR LST))  T )
          (     T       (POISK (CDR LST))   )
   )
))
```

Problem 12.

```
; ----------------------------------------------- ;
; Define a function (CDNR NL) that is equivalent ;
; to a function (CDDD...DR L) that has ;
; N letters D in its name ;
; -------------------------------------------------- ;
(DEFUN CDNR (LAMBDA (N LST)
   (COND ( (EQ N 0) LST )
         ( (EQ N 1) (CDR LST) )
         (  T  (CDNR (- N 1) (CDR LST)) )
   )
))
```

Problem 13.

```
; ---------------------------------------- ;
; Define a function that removes
every third element from a list ;
; ---------------------------------------- ;
(DEFUND FOR (LAMBDA (L))
   (COND ( (NULL L) NIL )
         ( (<   (LENGTH L) 3) L )
         (  T   (CONS (CAR L)
                      (CONS (CADR L)
                                (FO (CDR (CDDR L)))))
         )
   )
))
```

Task 14.

```
; -------------------------------------------------- ;
; Find the maximum element among the negative ;
; elements of a numeric list ;
; -------------------------------------------------- ;
( DEFUN MAIN (LAMBDA ( LST ) )
   (MAX (LST NEGATIVE))
))
; ------------------------- ;
(NEGATIVE DEFUN (LAMBDA (LST)
; Constructing a list made up of negative ;
; elements of the list LST ;
   (COND ( (NULL LST) NIL )
         ( (MINUSP (CAR LST))
                 (CONS (CAR LST)
                       (NEGATIVE (CDR LST))) )
         ( T (NEGATIVE (CDR LST)) )
   )
))
; ----------------------- ;
(DEFUN MAX (LAMBDA (LST)
; Determining the largest element of a single-level ;
; numeric list LST ;
   (COND ( (NULL LST) NIL )
         ( (EQ (LENGTH LST) 1) (CAR LST) )
         (  T  (MAX2 (CAR LST) (MAXIM (CDR LST))) )
   )
))
; ---------------------- ;
(DEFUN MAX2 (LAMBDA (X Y)
   (( (> X Y) X ) Y )
))
```

Task 15.

```
; ------------------------------------------------ ;
; Write a function REVERS that should join ;
; lists, where both lists should be reversed ;
; ------------------------------------------------ ;
(DEFUN REVERSE (LAMBDA (L1 L2)
   (APPEND (REVERSE L1) (REVERSE L2))
))
; ----------------------- ;
( DEFUN APPEND (LAMBDA ( X Y ) )
; Creates a list from the elements of lists X and Y ;
   (COND ( (NULL X) Y )
         (  T  (CONS (CAR X) (APPEND (CDR X) Y)) ))
))
```

Problem 16.

```
; ------------------------------------------------- ;
; Define a function EQUALL such that (EQUALL L) ;
; returns T if L is a list whose elements (at the ;
; topmost level) are all equal to each other and NIL - ;
; otherwise ;
; ------------------------------------------------- ;
(DEFUN EQUALL (LAMBDA (LST)
   (COND ( (EQ (LENGTH LST) 1) T )
         (   T   (AND (EQ (CAR LST) (CADR LST))
                          (EQUALL (CDR LST))
                 )
         )
   )
))
```

Problem 17.

```
; ------------------------------------------------- ;
; Define a function DOMINATE with two numeric lists
as arguments, such that it ;
; returns T if each atom of L is greater than the corresponding one
of M and NIL otherwise ;
; ------------------------------------------------- ;
(DEFUN DOMINATE (LAMBDA (L M)
   (COND ( (NULL L) T )
         (    T   (AND (  >      (CAR L) (CAR M))
                       (DOMINATE (CDR L) (CDR M)) ) )
   )
))
```

Problem 18.

```
; ---------------------------------------------------- ;
; Write a function INDEXMAX that returns the index ;
; of the largest number in a list of numeric atoms that has no
sublists ;
; ---------------------------------------------------- ;
(DEFUN INDEXMAX (LAMBDA (LST)
   (INDEX (MAXIM LST) LST)
))
; ------------------------ ;
(DEFUN INDEX (LAMBDA (X LST)
; Find the "location" of element X in the list LST ;
   (COND ( (NULL LST) 0)
          ( (EQ (CAR LST) X) 1 )
          (   T  (+ 1 (INDEX X (CDR LST))) )
   )
))
; ---------------------- ;
(DEFUN MAX (LAMBDA (LST)
; Determining the largest element of a numeric list LST ;
   (COND ( (NULL LST) NIL )
         ( (EQ (LENGTH LST) 1)  (CAR LST) )
         (   T  (MAX2 (CAR LST) (MAXIM (CDR LST))) )
   )
))
; --------------------- ;
(DEFUN MAX2 (LAMBDA (X Y)
   (COND ( (> X Y) X )
         (  T   Y )
   )
))
```

Problem 19.

```
; ------------------------------------------------------ ;
; Write a function (INRANGE V LST) that returns T if ;
; V lies between MAXIMUM and MINIMUM, including the bounds, and ;
; NIL otherwise. V is assumed to be a
numeric atom. MAXIMUM and MINIMUM are the largest and smallest ;
; elements of the given list ;
; ----------------------------------------------------- ;
( DEFUN INRANGE (LAMBDA ( V LST ) )
   (COND ( (AND ( <  V (MAXIMUM LST))
                ( >  V (MINIMUM LST)))  T )
         ( T  NIL )
   )
))
; ------------------------ ;
(DEFUN MAXIMUM (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (EQ (LENGTH LST) 1)  (CAR LST) )
         (   T  (MAX2 (CAR LST) (MAXIMUM (CDR LST))) )
   )
))
; --------------------- ;
(DEFUN MAX2 (LAMBDA (X Y)
   (COND ( (> X Y) X )
         (  T  Y )
   )
))
; -------------------------- ;
( DEFUN MINIMUM (LAMBDA ( LST )
   (COND ( (NULL LST) NIL )
         ( (EQ (LENGTH LST) 1) (CAR LST) )
         (  T  (MIN2 (CAR LST) (MINIMUM (CDR LST))) )
   )
))
; --------------------- ;
( DEFUN MIN2 (LAMBDA ( X Y ) )
   (COND ( (< X Y) X )
         ( T  Y )
   )
))
```

Task 20.

```
; ----------------------------------------------------- ;
; Build a function that removes from a numeric list ;
; all elements that match a given number and returns ;
; a list of the squares of the remaining elements ;
; ----------------------------------------------------- ;
(DEFUN MAN (LAMBDA) (X LST)
   (COPY (REMBER X LST))
))
; ------------------------- ;
(DEFUN REMEMBER(LAMBDA (X LST))
   (COND ( (NULL LST) NIL )
         ( (EQ X (CAR LST)) (REMBER X (CDR LST)) )
         (  T  (CONS (CAR LST) (REMBER X (CDR LST))) )
   )
 ))
; ------------------- ;
(DEFUN COPY (LAMBDA (L))
; The function returns a list consisting of squares ;
   (COND ( (NULL L) NIL )
         (  T  (CONS (* (CAR L) (CAR L))
                               (COPY (CDR L)) ) )
   )
))
```

Problem 21.

```
; -------------------------------------------- ;
; Determine if a numeric list contains two ;
; consecutive zero elements ;
; ------------------------------------------- ;
( DEFUN POISK (LAMBDA ( LST ) )
   (COND ( (NULL LST) NIL )
         ( (AND (EQ (CAR LST) (CAR (CDR LST)))
                (EQ (CAR LST) 0)) T )
         (  T   (POISK (CDR LST)) )
   )
))
```

In the next step we will continue to provide ***solutions to list processing problems*** .

Step 121.
Solved tasks. List processing tasks (end)

In this step we will continue ***to provide solutions to list processing problems*** .

Problem 22.

```
; ------------------------------------------------ ;
; Define a function LASTHALF whose value ;
; must be a list of the last n atoms in a list ;
; of 2n atoms. Thus, if X is (2 4 6 8), then ;
; the function (LASTHALF X) returns (6 8) ;
; ------------------------------------------------ ;
( DEFUN LASTHALF (LAMBDA ( LST ) )
   (COND ( (EQ (MOD (LENGTH LST) 2) 0)
                   (LASTN LST
                          (CAR (DIVIDE (LENGTH LST) 2))) )
         (  T  NIL  )
```

```
    )
))
; ------------------------- ;
(DEFUN FIRSTN (LAMBDA (LST N)
; Selects the first N elements of the list LST into the list;
   (COND ( (EQ N 1) (LIST (CAR LST)) )
         ( T (CONS (CAR LST)
                   (FIRSTN (CDR LST) (- N 1))))
   )
))
; ------------------------ ;
( DEFUN LASTN (LAMBDA ( LST N ) )
; Selects the last N elements of the list LST into the list;
   (REVERSE (FIRSTN (REVERSE LST) N))
))
```

Problem 23.

```
; -------------------------------------------------- ;
; Check if two lists are "congruent" ;
; (have the same structure) ;
; First solution ;
; -------------------------------------------------- ;
(DEFUN EQ_TYPE (LAMBDA (LST1 LST2)
   (COND
       ( (NOT (EQ (LENGTH LST1) (LENGTH LST2))) NIL )
       ( (AND (NULL LST1) (NULL LST2)) T )
       ( (AND (ATOM (CAR LST1)) (ATOM (CAR LST2)))
                  (EQ_TYPE (CDR LST1) (CDR LST2)) )
       ( (AND (NOT (ATOM (CAR LST1)))
              (NOT (ATOM (CAR LST2))))
       (AND (EQ_TYPE (CAR LST1) (CAR LST2))
                  (EQ_TYPE (CDR LST1) (CDR LST2))) )
       ( T NIL )
   )
))
; -------------------------------------------------- ;
; Check if two lists are "congruent" ;
; (have the same structure) ;
; Second solution ;
; -------------------------------------------------- ;
( DEFUN MAIN (LAMBDA ( LST1 LST2 ) )
   (EQUAL (COPY LST1) (COPY LST2))
))
; ----------------------- ;
(DEFUN COPY (LAMBDA (EXPN)
; The COPY function returns a copy of the structure ;
; of its argument ;
   (COND ( (ATOM EXPN) 1 )
         (   T  (CONS (COPY (CAR EXPN))
                      (COPY (CDR EXPN))) )
   )
))
```

Problem 24.

```
; -------------------------------------------- ;
; Find the sum of the numbers of a given letter in a word ;
; entered from the keyboard ;
; ------------------------------------------- ;
( DEFUN MAIN (LAMBDA ()
   (PRIN1 "Enter a word...")
```

```
    (SETQ F (UNPACK (READ)))
    (PRIN1 "Enter letter...")
    (SUMMA (READ)
           (PAIRLIS F (REVERSE (NUMER (LENGTH F))) NIL))
))
; ------------------------ ;
( DEFUN SUM (LAMBDA ( X LST ) )
    (COND ( (NULL LST) 0 )
          ( (EQ X (AAR LST))
                  (+ (CADR (CAR LST))
                     (SUMMA X (CDR LST))) )
          (  T  (SUMMA X (CDR LST)) )
    )
))
; --------------------- ;
(DEFEN NUMBER (LAMBDA (N)
; Building a list (1 2 3 ... N) ;
    (COND
       ( (EQ N 0) NIL )
       (    T     (CONS N (NUMER (- N 1)))   )
    )
))
; ------------------------------------ ;
(DEFUN PAIRLIS) (LAMBDA) (KEY DATA ALIST)
; Constructing an A-list from a list of keys KEY and a list of ;
; data DATA by adding new pairs to the
existing list ALIST ;
    (COND ( (NULL KEY)  ALIST )
          ( (NULL DATA) ALIST )
          (  T  (CONS (CONS (CAR KEY) (CAR DATA))
                      (PARLY
                         (CDR KEY) (CDR DATA) ALIST) )
          )
    )
))
```

Problem 25.

```
; ------------------------------------------------ ;
; Write a function that removes duplicates ;
; of elements from a list, for example: ;
; (A B D D A) --> (A B D) ;
; ----------------------------------------------- ;
(DEFUN LIST-SET (LAMBDA (LST)
; The LIST-SET function converts an list LST into a set ;
    (COND
       ( (NULL LST) NIL )
       ( (MEMBER (CAR LST) (CDR LST))
                   (LIST-SET (CDR LST)) )
       (    T    (CONS (CAR LST) (LIST-SET (CDR LST))) )
    )
))
```

Problem 26.

```
; --------------------------------------------------- ;
; Write a function TOP such that (TOP L) returns ;
; the maximum depth of sublists of list L. ;
; Thus, if L is (1 4 (2 6 (3 7) 8)), then (TOP L) ;
; returns 3 ;
; --------------------------------------------------- ;
(DEFUN TOP (LAMBDA (LST)
   (COND
      ( (ALLATOMP LST) 1 )
      (  T  (MAX (+ 1 (TOP (CAR LST)))
                 (TOP (CDR LST))) )
   )
))
; -------------------- ;
(DEFUN MAX (LAMBDA (M N))
; The MAX function returns the larger of two numbers ;
   (COND ( (> M N) M )
         (  T      N )
   )
))
; ------------------------- ;
(DEFUN ALLTOMP (LAMBDA (LST))
; A predicate that allows us to determine whether ;
; all elements of an list LST are atoms ;
   (COND
       ( (NULL LST)  T                                  )
       (  T (AND (ATOM (CAR LST)) (ALLATOMP (CDR LST))) )
   )
))
```

Problem 27.

```
  ; --------------------------------------- ;
  ; In the list, swap the first and last ;
  ; elements ;
  ; --------------------------------------- ;
  (DEFUN PERESTANOVKA (LAMBDA (LST)
     (COND ( (NULL LST) NIL )
           ( (EQ (LENGTH LST) 1) LST )
           ( T  (REVERSE
                   (CONS (CAR LST)
                   (REVERSE (CONS (CAR (REVERSE LST))
                   (REVERSE (CDR (REVERSE (CDR LST)))))))
           )
     )
  ))
```

Problem 28.

```
  ; ----------------------------------------------- ;
  ; Define the DISTL function (distribution to the left), ;
  ; the action of which we will consider using an example: ;
  ; (A (B C ... D)) --> ((A B) (A C) ... (A D)) ;
  ; ----------------------------------------------- ;
  (DEFUN DISTL (LAMBDA (LST)
     (AAA (CAR LST) (CADR LST))
  ))
  ; ---------------------- ;
  (DEFUN AAA(LAMBDA (X LST))
     (COND ( (NULL LST) NIL)
           (    T   (CONS (LIST X (CAR LST))
                          (AAA X (CDR LST))) )
     )
  ))
```

Problem 29.

```
; ----------------------------------------------- ;
; Define the DISTR function (distribution on the right), ;
; the action of which we will consider using an example: ;
; ((A B ... H) C) --> ((A C) (B C) ... (H C)) ;
; ----------------------------------------------- ;
(DEFUN DISTR (LAMBDA (LST)
   (AAA (CAR LST) (CADR LST))
))
; ----------------------- ;
(DEFUN AAA(LAMBDA (LST))
   (COND ( (NULL LST) NIL )
         (   T  (CONS (LIST (CAR LST) X)
                      (AAA (CDR LST) X)) )
   )
))
```

Problem 30.

```
; ---------------------------------------------- ;
; Describe the function SP that rewrites all ;
; elements not equal to X to the beginning, and equal to X - ;
; to the end of the given list ;
; -------------------------------------------- ;
(DEFUN SP (LAMBDA (X LST)
   (COND ( (NULL LST) NIL )
         (  T   (APPEND (REMBER X LST)
                        (COPY X (KOL X LST))) )
   )
))
; ----------------------- ;
(DEFUN CALL(LAMBDA (X LST))
; Count the number of times an element X ;
; appears in an list LST ;
   (COND ( (NULL LST) 0 )
         ( (EQ (CAR LST) X)
              (+ (KOL X (CDR LST)) 1) )
         (  T  (KOL X (CDR LST))    )
   )
))
; -------------------------- ;
(DEFUN REMEMBER(LAMBDA (X LST))
   (COND ((NULL LST) NIL)
         ( (EQ X (CAR LST)) (REMBER X (CDR LST)) )
         (  T   (CONS (CAR LST)
                      (REMBER X (CDR LST))) )
   )
))
; ---------------------- ;
(DEFUN COPY (LAMBDA (X N)
; A function that allows you to copy a given ;
; element into a list N times.
   (COND ( (EQUAL N 0) NIL )
         (  T  (CONS X (COPY X (- N 1))) )
   )
))
```

Problem 31.

```
; ---------------------------------------------------- ;
; Find in the list an atom that occurs ;
; one atom before the given one ;
; ---------------------------------------------------- ;
```

```
(DEFUN PATOM (LAMBDA (X LST)
   (COND ( (NULL LST) NIL )
         ( (< (LENGTH LST) 3) NIL )
         (  T  (NTH (- (NATOM X LST) 2)
                    LST) )
    )
))
; ---------------------- ;
(DEFUN NTH (LAMBDA (N LST)
; The function returns the N-th element of the list LST ;
   (COND ( (< N 1) 0 )
         ( (EQ N 1) (CAR LST) )
         (   T   (NTH (- N 1) (CDR LST)) )
   )
))
; ------------------------ ;
(DEFUN NATO (LAMBDA (10 LST)
; The function returns the position of the specified element ;
   (COND ( (NULL LST) 0)
         ( (EQ X (CAR LST)) 1)
         ( (MEMBER X LST) (+ 1 (NATOM X (CDR LST))) )
         ( T  0 )
   )
))
```

Problem 32.

```
; -------------------------------------------- ;
; Write a function INSLIST that depends on three ;
; arguments N, U, and V, inserting into a list U, ;
; starting with the N-th element, a list V ;
; ------------------------------------------- ;
(DEFUN MAIN (LAMBDA) (LST1 N LST2)
   (COND ( (EQ (LENGTH LST1) N) (APPEND LST1 LST2) )
         (  T  (APPEND (FIRSTN LST1 N)
                       (APPEND LST2
                              (LASTN LST1
                                     (-
                                        (LENGTH LST1) N)))
                )
         )
   )
))
; ------------------------- ;
(DEFUN FIRSTN (LAMBDA (LST N)
; Selects the first N elements of the list LST into the list;
   (COND ( (EQ N 1) (LIST (CAR LST)) )
         (  T  (CONS (CAR LST)
                     (FIRSTN (CDR LST)
                             (- N 1))))
   )
))
; ------------------------ ;
( DEFUN LASTN (LAMBDA ( LST N ) )
; Selects the last N elements of the list LST into the list;
   (REVERSE (FIRSTN (REVERSE LST) N))
))
; ----------------------------- ;
(DEFUN APPEND (LAMBDA (LST1 LST2)
; Creates a list from the first N elements of the list LST1 ;
; and the inserted list LST2 ;
   (COND ( (NULL LST1) LST2 )
         (   T  (CONS (CAR LST1)
                      (APPEND (CDR LST1) LST2) ))
```

```
      )
   ))
```

Problem 33.

```
   ; ----------------------------------------------------- ;
   ; Make a list congruent to the given one and consisting of ;
   ; depths of immersion of each element of the original list ;
   ; ----------------------------------------------------- ;
   (DEFUN AA(LAMBDA (LST))
      (AA1 1 LST)
   ))
   ; ---------------------- ;
   (DEFUN AA1 (LAMBDA (N LST))
      (COND ( (NULL LST) NIL)
            (  T  (CONS (GETN N (CAR LST))
                        (AA1 N (CDR LST))) )
      )
   ))
   ; ---------------------- ;
   (DEFUN GETN(LAMBDA (N LS)
      (COND   ( (ATOM LS) N )
             ( T (COND ( (< (LENGTH LS) 2)
                          (LIST (GETN (+ 1 N)
                                      (CAR LS))) )
                        ( T  (AA1 (+ 1 N) LS) )
                 )
             )
      )
   ))
```

Problem 34.

```
   ; ------------------------------------------------- ;
   ; Function, converts a list of numbers into a list of numbers that ;
   ; occur more than once in it ;
   ; ------------------------------------------------- ;
   (DEFUN KOLAM (LAMBDA (LST)
      (EL (LIST-SET (TH LST LST)))
   ))
   ; ------------------- ;
   (DEFUN EL (LAMBDA (LST)
      (COND
         ( (NULL LST) NIL )
         ( (>; (CADR (CAR LST)) 1)
              (CONS (CAR (CAR LST)) (EL (CDR LST))) )
         (  T  (EL (CDR LST)) )
      )
   ))
   ; ------------------------ ;
   (DEFUN LIST-SET (LAMBDA (LST)
   ; The LIST-SET function converts an list LST into a set ;
      (COND ( (NULL LST) NIL )
            ( (MEMBER (CAR LST) (CDR LST))
                 (LIST-SET (CDR LST)) )
            (  T  (CONS (CAR LST) (LIST-SET (CDR LST))) )
      )
   ))
   ; ----------------------- ;
   ( DEFUN TH (LAMBDA ( LST LST0 )
   ; Construct a new list containing sublists of the form ;
   ; (Element Number_of_repetitions_of_this_element_in_list) ;
```

```
    (COND ( (NULL LST) NIL)
          (  T  (CONS (LIST (CAR LST)
                            (KOL (CAR LST) LST0))
                      (TH (CDR LST) LST0)) )
    )
))
; ---------------------- ;
(DEFUN CALL (LAMBDA (M LST))
; Count the number of repetitions of element M in the list LST ;
    (COND ( (NULL LST) 0 )
          ( (EQ (CAR LST) M)
               (+ (KOL M (CDR LST)) 1) )
          (  T  (KOL M (CDR LST)) )
    )
))
```

Problem 35.

```
; ----------------------------------------------------- ;
; Check if this sentence contains words that ;
; are "reversals" of the first word of the sentence ;
; ----------------------------------------------------- ;
( DEFUN MAIN (LAMBDA ( LST ) )
    (COND ( (NULL LST) NIL )
          (  T  (OR (WWWW (CAR LST)
                          (COPYINVERT (CDR LST)))
                    (MAIN (CDR LST))
                )
          )
    )
))
; ----------------------- ;
( DEFUN WWWW (LAMBDA ( X LST ) )
    (COND ( (NULL LST) NIL )
          ( (MEMBER X LST) T )
          (   T  (WWWW (CDR LST)) )
    )
))
; --------------------------- ;
(DEFUN COPYINVERT (LAMBDA (LST)
    (COND ( (NULL LST) NIL )
          ( T (CONS (PACK (REVERSE (UNPACK (CAR LST))))
                      (COPYINVERT (CDR LST))
              )
          )
    )
))
```

Problem 36.

```
; --------------------------------------------------- ;
; Write a program to convert integer N into an integer, ;
; the notation of which differs from the notation of number N by
the permutation of the first and last digits ;
; --------------------------------------------------- ;
(DEFUN FOR (LAMBDA (N)
    (SETQ L (UNPACK N))
    (COND ( (< N 10) N )
          ( T (NUMBER (STRING-NUM
                         (APPEND
                            (LIST (CAR (REVERSE L)))
                            (REVERSE
                            (CDR (REVERSE (CDR L))))
                            (CAR L))
                     )
              )
           )
      )
))
; ------------------------- ;
(DEFUN STRING-NUM(LAMBDA (X))
; Translate a list of digits into a list containing ;
; the corresponding single-digit numbers ;
    (COND ( (NULL X) NIL )
          (  T  (CONS (POSITION (CAR X) (UNPACK 123456789))
                      (STRING-NUM (CDR X))
                )
          )
    )
))
; ---------------------------- ;
(DEFUN POSITION (LAMBDA (X LST)
; The POSITION function returns the position of the atom X ;
; in a single-level list LST (the first element has ;
; number 1). If the element is not in the list, the function ;
; returns 0 ;
    (COND ( (NULL LST)       0 )
          ( (EQ X (CAR LST)) 1 )
          ( (MEMBER X LST)
               (+ 1 (POSITION X (CDR LST))) )
          ( T  0 )
    )
))
; ----------------------- ;
(DEFUN NUMBER (LAMBDA (LST))
; Given a numeric list LST containing single-digit numbers. ;
; "Build" an integer from the elements of this list ;
  (COND ( (NULL LST) 0 )
           (  T  (+ (* (CAR LST)
                         (STAGE 10
                              (- (LENGTH LST) 1))
                    )
                    (NUMBER (CDR LST))
               )
           )
  )
))
; ----------------------- ;
( DEFUN STEP (LAMBDA ( X A )
; Raising an integer X to a non-negative integer ;
; power A ;
    (COND ( (ZEROP A) 1 )
          ( (ZEROP (- A 1)) X )
          (  T  (* (STEPEN X (- A 1)) X) )
    )
))
```

Problem 37.

```
  ; ------------------------------------------------ ;
  ; Try to restore the condition of this problem! ;
  ; ------------------------------------------------ ;
  (DEFUN AAA (LAMBDA (AND LST))
     (SETQ W (COPY LST LST Y))
     (COPY1 WW)
  ))
```

```
; ---------------------------- ;
(DEFUN COPY (LAMBDA (LST LST1 Y)
; Construct a list congruent to the list LST and ;
; containing symbols of the form G*** instead of occurrences of ;
; symbols Y ;
   (COND ( (NULL LST) NIL )
         ( (AND (ATOM LST) (EQ LST Y))  (GENSYM) )
         ( (AND (ATOM LST) (NOT (EQ LST Y))) LST )
         (  T  (CONS (COPY (CAR LST) LST1 Y)
                     (COPY (CDR LST) LST1 Y)
               )
         )
   )
))
; --------------------------- ;
(DEFUN COPY1 (LAMBDA (LST LST1)
; Construct a list congruent to the list LST and ;
; containing the "depths of immersion" of elements in the ;
; list LST instead of G*** symbols ;
   (COND ( (NULL LST) NIL )
         ( (AND (ATOM LST) (EQ (CAR (UNPACK LST)) G))
                                    (POISON LST LST1) )
         ( (AND (ATOM LST) (NOT (EQ (CAR (UNPACK LST)) G)))
                                    LST )
         (  T  (CONS (COPY1 (CAR LST) LST1)
                     (COPY1 (CDR LST) LST1)) )
   )
))
; ----------- ;
(SETQ GENSYM 0)
(DEFUN GENSYM (LAMBDA (NUM LST)
   (SETQ NUM (- 4 (LENGTH GENSYM)))
   (LOOP
      ( (ZERO NUMBER) )
      ( PUSH 0 LST )
      ( SETQ NUM (- NUM 1)) )
   (PROG1
      (PACK (NCONC (CONS (QUOTE G) LST) (LIST GENSYM)))
      (SETQ GENSYM (+ GENSYM 1))
   )
))
; ------------------------ ;
( DEFUN POISK (LAMBDA ( X LST ) )
; Finding the "depth of immersion" of X in the list LST ;
   (COND ( (MEMBER X LST) 1 )
         ( (MEMBER2 X (CAR LST))
              (+ 1 (POISK X (CAR LST))) )
         (  T  (POISK X (CDR LST)) )
   )
))
; -------------------------- ;
(DEFUN MEMBER2 (LAMBDA (X LST)
; The MEMBER2 predicate establishes the occurrence ;
; of element X in the multilevel list LST ;
   (COND ( (NULL LST) NIL)
         (  T  (OR (COND ( (ATOM (CAR LST))
                              (EQ X (CAR LST)) )
                         (  T  (MEMBER2 X (CAR LST)) )
                   )
                   (MEMBER2 X (CDR LST))) )
   )
))
```

In the next step we ***will introduce string processing tasks*** .

Step 122.
Solved tasks. String processing tasks

In this step we will introduce ***string processing tasks*** .

Task 1.

```
; ---------------------------------------------------- ;
; Write down the algorithm for deleting from the word X all the letters ;
; standing in odd places after the letter "A" ;
; ---------------------------------------------------- ;
(DEFUN PACDEL (LAMBDA ()
   (SETQ LST (UNPACK (READ)))
   (COND ( (< (LENGTH LST) 2) (PACK LST) )
         (  T  (PACK (CONS (CAR LST)
                           (DELAP_A (CDR LST))))
         )
   )
))
; ------------------------ ;
(DEFUN EIGHT_A (LAMBDA (LST)
   (COND ( (< (LENGTH LST) 2) LST )
         ( (EQ (CAR LST) A)
                  (CONS (CAR LST)
                        (DELAP_A (CDDR LST))) )
         (   T    (CONS (CAR LST)
                        (CONS (CADR LST)
                              (DELAP_A (CDDR X))))
         )
   )
))
```

Task 2.

```
; --------------------------------------------------- ;
; Create an algorithm that highlights all the letters
"O" in the word X with the symbols "#", for example, the word "SOSNA" should ;
; be transformed into the word S#O#SNA" ;
; --------------------------------------------------- ;
(DEFUN ILLUSTRATED (LAMBDA ()
   (PACK (PRIMER (UNPACK (READ))))
))
; ----------------------- ;
(DEFUN FIRST (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (EQ (CAR LST) O)
                 (CONS # (CONS O (CONS #
                                   (PRIMER (CDR LST)))))
         )
         (  T  (CONS (CAR LST) (PRIMER (CDR LST))) )
   )
))
```

Task 3.

```
; ------------------------------------------------- ;
; In a given sentence, find words of a given length ;
; -------------------------------------------------- ;
(DEFUN MAXIM (LAMBDA (N LST)
   (COND ( (NULL LST) NIL )
         ( (MAX2 (CAR LST))
               (CONS (CAR LST) (MAXIM N (CDR LST))) )
         (  T  (MAXIM N (CDR LST)) )
   )
))
; -------------------- ;
(DEFUN MAX2 (LAMBDA (Y))
   (COND ( (EQ (LENGTH (UNPACK Y)) N ) T )
         (  T  NIL )
   )
))
```

Task 4.

```
; ------------------------------------------------- ;
; Write an algorithm that finds out if the word X ;
; contains the letter "R", and if so, replace it with "C" ;
; ------------------------------------------------- ;
(DEFUN A (LAMBDA ()
   (PACK (AF (UNPACK (READ))))
))
; -------------------- ;
(DEFUN OF (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (EQ (CAR LST) R)
                 (CONS C (AF (CDR LST))) )
         (   T   (CONS (CAR LST) (AF (CDR LST))) )
   )
))
```

Task 5.

```
; -------------------------------------------------- ;
; Write down the algorithm for counting the number of letters "O" standing ;
; in a word in places whose number is a multiple of 4 ;
; -------------------------------------------------- ;
( DEFUN MAIN (LAMBDA ()
   (PLL (UNPACK (READ)))
))
; --------------------- ;
(DEFUN PLL (LAMBDA (LST)
   (COND  ( (< (LENGTH LST) 4) 0 )
          ( (EQ (CAR (CDDDR LST)) O)
                    (+ 1 (PLL (CDR (CDDDR LST))) )
          )
          (   T    (PLL (CDR (CDDDR LST))) )
   )
))
```

Task 6.

```
;--------------------------------------------------- ;
; Write an algorithm that finds out if the word X ;
; contains the letter "A" in an even place after the letter "K" ;
;--------------------------------------------------- ;
(DEFUN A (LAMBDA ()
   (RR (UNPACK (READ)))
))
; -------------------- ;
( DEFUN RR (LAMBDA ( LST ) )
   (COND ( (< (LENGTH LST) 2) NIL )
         ( (AND (EQ (CAR LST) K) (EQ (CADR LST) A)) T )
         (  T  (RR (CDDR LST)) )
   )
))
```

Task 7.

```
; -------------------------------------------------;
; Write an algorithm that checks if the word X ;
; contains two identical letters ;
; ------------------------------------------------ ;
( DEFUN MAIN (LAMBDA ()
   (TWOEQ (UNPACK (READ)))
))
; ----------------------- ;
(DEFUN TWOEQ(LAMBDA (LST))
   (COND
      ( (NULL LST)                    NIL              )
      ( (MEMBER (CAR LST) (CDR LST))  T                )
      (  T                           (TWOEQ (CDR LST)) )
   )
))
```

Task 8.

```
; -------------------------------------------------- ;
; Delete from the given word all letters whose number
in the word is a multiple of 4 ;
; -------------------------------------------------- ;
 (DEFUN F(LAMBDA (L))
   (FO (UNPACK L))
 ))
 ; ----------------- ;
 (DEFUN FO(LAMBDA (L) )
   (COND ( (NULL L) NIL )
         ( (< (LENGTH L) 4 )  L )
         (  T   (CONS (CAR L)
                      (CONS (CADR L)
                          (CONS (CADDR L)
                                (FO (CDDR (CDDR L))))))
         )
   )
))
```

Task 9.

```
; ------------------------------------------------ ;
; Write a program to count the number of combinations ;
; "MU" in a given word ;
; ------------------------------------------------ ;
(DEFUN UU (LAMBDA ()
   (MU (UNPACK (READ)))
))
; -------------------- ;
(DEFUN IN(LAMBDA (LST))
   (COND ( (< (LENGTH LST) 2) 0 )
         ( (AND (EQ (CAR LST) M) (EQ (CADR LST) U))
               (+ 1 (MU (CDR LST))) )
         (  T  (MU (CDR LST)) )
   )
 ))
```

Task 10.

```
 ; ------------------------------------------ ;
 ; Replace the letter "A" in the word with the combination "KU" ;
 ; ------------------------------------------ ;
 ( DEFUN MAIN (LAMBDA ()
    (PACK (INSDEL (UNPACK (READ))))
 ))
 ; ---------------------- ;
 (DEFUN INSDEL(LAMBDA (LST))
    (COND ( (NULL LST) NIL )
          ( (EQ (CAR LST) A)
                (CONS K (CONS U (INSDEL (CDR LST)))) )
          (   T   (CONS (CAR LST) (INSDEL (CDR LST)))  )
    )
 ))
```

Task 11.

```
 ; ---------------------------------------------- ;
 ; Write an algorithm for checking whether the first ;
 ; letter of word X matches the last letter of word Y ;
 ; ---------------------------------------------- ;
 (DEFUN PROVFL(LAMBDA (X Y))
    (COND ( (EQ (CAR (UNPACK X))
                (CAR (REVERSE (UNPACK Y)))) T )
          (  T  NIL  )
    )
 ))
```

Problem 12.

```
 ; ----------------------------------------------- ;
 ; Find out if the given word is a reversal ;
 ; ----------------------------------------------- ;
 (DEFUN FOR (LAMBDA (W)
    (COND ( (EQUAL (UNPACK W) (REVERSE (UNPACK W))) T )
          (   T   NIL  )
    )
 ))
```

Problem 13.

```
 ; ----------------------------------------- ;
 ; Create an algorithm that checks how many times ;
 ; the first letter of the word X occurs in the word Y ;
 ; ----------------------------------------- ;
 ( DEFUN MAIN (LAMBDA ( X Y ) )
    (RR (CAR (UNPACK X)) (UNPACK Y))
 ))
 ; -------------------- ;
 ( DEFUN RR (LAMBDA ( Z LST ) )
    (COND ( (NULL LST)  0 )
          ( (EQ (CAR LST) Z) (+ 1 (RR Z (CDR LST))) )
          (   T  (RR Z (CDR LST)) )
    )
 ))
```

Task 14.

```
; -------------------------------------------------- ;
; Create an algorithm for calculating the total number of ;
; letters "M" and "N" in the word Y ;
; -------------------------------------------------- ;
(DEFUN MAIN) (LAMBDA (Y)
   (+ (BUK M (UNPACK Y))
      (BUK N (UNPACK Y)))
))
; ---------------------- ;
(DEFUN BOOK (LAMBDA (X LST))
   (COND ( (NULL LST) 0                            )
         ( (EQ (CAR LST) X) (+ 1 (BUK X (CDR LST))) )
         (        T         (BUK X (CDR LST))       )
   )
))
```

Task 15.

```
; ----------------------------------------------------- ;
; Write a program that will delete the letter "B" ;
;  (both uppercase and lowercase) from a given word ;
; ----------------------------------------------------- ;
( DEFUN V (LAMBDA ()
   (PACK (REMOVE B (REMOVE b (UNPACK (READ)))))
))
; --------------------------- ;
(DEFUN REMOVE (LAMBDA (ATM LST)
; Returns a list with all occurrences of ;
; ATM in the list LST removed ;
   (COND ( (NULL LST) NIL )
         ( (EQ ATM (CAR LST))
                     (REMOVE ATM (CDR LST)) )
         (   T   (CONS (CAR LST) (REMOVE ATM (CDR LST))) )
   )
))
```

Problem 16.

```
; ------------------------------------------------------- ;
; Write an algorithm that finds out how many times the second letter ;
; of the word LST1 occurs in the word LST2 in even places ;
; ------------------------------------------------------- ;
(DEFUN CALL (LAMBDA (LST1 LST2))
   (COND ( (NULL LST2) 0 )
         ( (EQ (CADR LST1) (CADR LST2))
                 (+ 1 (KOL LST1 (CDDR LST2))) )
         (   T   (KOL LST1 (CDDR LST2)))
   )
))
```

Problem 17.

```
; ------------------------------------------------ ;
; Write an algorithm for checking whether a word contains the letter ;
; "P". If so, find the number of the last of them ;
;------------------------------------------------- ;
(DEFUN MAIN (LAMBDA ()
   (SETQ LST (UNPACK (READ)))
   (COND ( (MEMBER P LST)
               (+ 1 (-
                      (LENGTH LST)
                      (POSITION P (REVERSE LST)))) )
         (   T   NIL  )
   )
))
; --------------------------- ;
(DEFUN POSITION (LAMBDA (X LST)
   (COND   ( (NULL LST)        0)
           ( (EQ X (CAR LST)) 1)
           ( (MEMBER X LST)
                (+ 1 (POSITION X (CDR LST))) )
           (  T   0  )
   )
))
```

Problem 18.

```
; ---------------------------------------------------- ;
; Find the word of maximum length in a given sentence ;
; ---------------------------------------------------- ;
(DEFUN MAX (LAMBDA (LST)
   (COND ( (NULL LST) NIL )
         ( (EQ (LENGTH LST) 1) (CAR LST) )
         (   T   (MAX2 (CAR LST) (MAXIM (CDR LST)))  )
   )
))
; ---------------------- ;
(DEFUN MAX2 (LAMBDA (X Y)
   (COND ( (> (LENGTH (UNPACK X))
              (LENGTH (UNPACK Y)))  X )
         (  T   Y  )
   )
))
```

Problem 19.

```
; -------------------------------------------------- ;
; Write a program that will determine how many ;
; different symbols of the word A are used in the spelling
of A more than once ;
; -------------------------------------------------- ;
(DEFUN COLUMN (LAMBDA (A)
   (OTO (LIST-SET (TH (UNPACK A) (UNPACK A))))
))
; ------------------------ ;
( DEFUN TH (LAMBDA ( LST LST0 )
; Construct a new list containing elements of the form ;
; (Element Number_of_repetitions_of_this_element_in_list) ;
   (COND ( (NULL LST) NIL)
         (  T  (CONS (LIST (CAR LST)
                           (KOL (CAR LST) LST0))
                     (TH (CDR LST) LST0)) ))
))
; ---------------------- ;
(DEFUN CALL (LAMBDA (M LST))
; Count the number of repetitions of element M in the list LST ;
   (COND ( (NULL LST) 0 )
         ( (EQ (CAR LST) M)
              (+ (KOL M (CDR LST)) 1) )
         (  T  (KOL M (CDR LST)) ))
))
; ------------------------- ;
(DEFUN LIST-SET (LAMBDA (LST)
; The function converts the list LST into a set ;
   (COND ( (NULL LST) NIL )
         ( (MEMBER (CAR LST) (CDR LST))
             (LIST-SET (CDR LST)) )
         (   T   (CONS (CAR LST) (LIST-SET (CDR LST))) )
   )
))
; -------------------- ;
(DEFUN OTO (LAMBDA (LST)
   (COND ( (NULL LST) 0 )
         ( (> (CADAR LST) 1)
                 (+ 1 (OTO (CDR LST))) )
         (  T    (OTO (CDR LST)) )
   )
))
```

Task 20.

```
; --------------------------------------------- ;
; From this sentence, select words that have ;
; a given number of letters (words in the sentence ;
; are separated by the symbol "_") ;
; --------------------------------------------- ;
(DEFUN W (LAMBDA (N WORD)
   (KUKU (MM N (UNPACK WORD)))
))
; --------------------- ;
( DEFUN MM (LAMBDA ( N LST ) )
   (COND ( (NULL LST) NIL )
         ( (EQ (- (POSITION _ LST) 1) N)
              (APPEND
                   (LIST (NTN N LST))
                   (MM N (MEM _ LST))) )
         (  T  (MM N (MEM _ LST)) )
   )
))
; ---------------------- ;
(DEFUN NTN (LAMBDA (N LST)
; The function returns the first N elements of the list LST ;
    (COND ( (EQ (LENGTH LST) N)
               LST)
          (  T  (NTN N (DR LST)) )
    )
))
; ------------------- ;
(DEFUN DR (LAMBDA (LST))
; The function removes the last element from the list LST ;
   (REVERSE (CDR (REVERSE LST)))
))
; --------------------------- ;
(DEFUN POSITION (LAMBDA (X LST)
; Returns the position of atom X in the list LST ;
   (COND ( (NULL LST)       0)
           ( (EQ X (CAR LST)) 1)
           ( (MEMBER X LST)
                (+ 1 (POSITION X (CDR LST)))   )
           ( T  0 )
      )
  ))
  ; ---------------------- ;
  (DEFUN MEM (LAMBDA (A LST)
  ; Returns the part of the list LST after the occurrence of element A ;
     (COND ( (NULL LST) NIL )
           ( (EQ (CAR LST) A) (CDR LST) )
           ( T (MEM A (CDR LST)) )
     )
  ))
  ; ----------------------- ;
  ( DEFUN APPEND (LAMBDA ( X Y ) )
     (COND ( (NULL X) Y )
           ( T (CONS (CAR X) (APPEND (CDR X) Y)) )
     )
  ))
  ; --------------------- ;
  (DEFUN CHICKEN (LAMBDA (LST)
     (COND ( (NULL LST) NIL )
           (  T  (CONS (PACK (CAR LST))
                       (KUKU (CDR LST))) )
     )
  ))
```

In the next step we ***will present problems that illustrate the imperative programming technology*** .

Step 123.
Solved problems. Problems to illustrate imperative programming in LISP

In this step we ***will present problems that illustrate the imperative style of programming in the* LISP** language .

Task 1.

```
  ; ---------------------------------------------------- ;
  ; Is there a natural number I not exceeding
  a given number N such that the number 2*I+15 will be ;
  ; composite ;
  ; ---------------------------------------------------- ;
  (DEFUN ZAD19 (LAMBDA (N))
     (SETQ I 1)
     (LOOP
        ( (> I N) PRIN1 "There is no such number!" )
        ( (SIMPLE (+ (* 2 I) 15)) I  )
        (SETQ I (+ I 1))
     )
  ))
  ; --------------------- ;
  (DEFUN SIMPLE (LAMBDA (N)
     (SETQ FLAG 1)
     (COND ( (AND (EQ (MOD N 2) 0) (NOT (EQ N 2))) NIL )
           (  T  ( (SETQ I 3)
                  (LOOP
                     ( (< (CAR (DIVIDE N 2)) I) )
                     ( (EQ (CDR (DIVIDE N I)) 0)
                          (SETQ FLAG 0) )
```

```
                      ( SETQ I (+ I 2) )
                   )
                   (COND ( (EQ FLAG 0)  NIL )
                         (   T          T   ))) )
     )
  ))
```

Task 2.

```
  ; --------------------------------------------- ;
  ; Determine what digit the number 7 ends with, ;
  ; raised to the power of 77 ;
  ; --------------------------------------------- ;
  (DEFUN WITH (LAMBDA (NUM1 NUM2))
     (CAR (REVERSE (UNPACK (POWER NUM1 NUM2))))
  ))
  ; ---------------------------- ;
  (DEFUN POWER (LAMBDA) (NUM1 NUM2)
      ( (AND (ZEROP NUM1) (MINUSP NUM2))
            (PRINT "Result does not exist!")
      (SETQ NUM3 1)
      (LOOP
         (SETQ NUM2 (DIVIDE NUM2 2))
         ( ( (EQ (CDR NUM2) 1)
                (SETQ NUM3 (* NUM1 NUM3)) ) )
         ( SETQ NUM2 (CAR NUM2) )
         ( (ZEROP NUM2) NUM3 )
         ( SETQ NUM1 (* NUM1 NUM1))
      )
  ))
```

Task 3.

```
  ; ------------------------------------------------- ;
  ; Add three digits to 523... so that the resulting
  six-digit number is ; ; divisible by 7, 8 and 9 ;
  ; ------------------------------------------------- ;
  (DEFUN OF (LAMBDA ()
     (SETQ N 523000)
       (LOOP
          ((EQ N 999999) 0)
          ((AND (ZEROP (CDR (DIVIDE N 7)))
                (ZEROP (CDR (DIVIDE N 8)))
                (ZEROP (CDR (DIVIDE N 9)))) N)
          (SETQ N (+ N 1))
       )
  ))
```

Task 4.

```
  ; ------------------------------------------------- ;
  ; Determine whether the number 210-p is prime if ;
  ; the number p is prime and 100<p<200 ;
  ; ------------------------------------------------- ;
  (DEFUN PROST (LAMBDA (P)
     (COND ( (AND (SIMPLE P)
                  (AND (> P 100) (< P 200))
             )
             (SIMPLE (- 210 P))
           )
```

```
    )
))
; ---------------------- ;
(DEFUN SIMPLE (LAMBDA (N)
; A predicate that allows one to determine whether ;
; a given integer N is prime ;
   (SETQ FLAG 1)
   (COND ( (AND (EQ (MOD N 2) 0) (NOT (EQ N 2))) PRIN1 "Not simple!" )
         (  T  ( (SETQ I 3)
                (LOOP
                   ( (< (CAR (DIVIDE N 2)) I) )
                   ( (EQ (CDR (DIVIDE N I)) 0)
                        (SETQ FLAG 0) )
                   ( SETQ I (+ I 2) )
                )
                (COND ( (EQ FLAG 0)  NIL )
                      (   T          T   ))) )
   )
))
```

Task 5.

```
; -------------------------------------------------- ;
; For what smallest natural number M is the number M^3+3^M ;
; divisible by 7 ;
; -------------------------------------------------- ;
(DEFUN FAY (LAMBDA (K)
   (SETQ M 1)
   ( LOOP
       ( (> M K) 0 )
       ( SETQ L (STEP 3 M) )
       ( SETQ N (STEP M 3) )
       ( SETQ T (+ L N)    )
       ( (ZEROP (CDR (DIVIDE T 7)))  M )
       ( SETQ M (+ M 1) )
   )
))
; ----------------------- ;
( DEFUN STEP (LAMBDA ( X A )
   (COND ( (ZEROP A) 1 )
         ( (ZEROP (- A 1))     X      )
         (  T  (* X (STEP X (- A 1))) )
   )
))
```

Task 6.

```
; ---------------------------------------- ;
; Find integers from 1 to N that are ;
; divisible by 3, 5, and 7 ;
; ---------------------------------------- ;
(DEFUN DA1 (LAMBDA ( N )
   (SETQ L 0)
   (SETQ I 1)
   ( LOOP
       ( (> I N) ((PRINT "Total numbers") L) )
       (COND ( (AND (ZEROP (CDR (DIVIDE I 3)))
                    (ZEROP (CDR (DIVIDE I 5)))
                    (ZEROP (CDR (DIVIDE I 7))))
                             ( (SETQ L (+ L 1)) (PRINT I))  ))
       (SETQ I (+ I 1))
   )
```

```
  ))
```

Task 7.

```
  ; ------------------------------------------- ;
  ; Are there 4 consecutive natural ;
  ; numbers whose sum of squares is equal to the sum of ;
  ; the squares of the next three natural numbers? ;
  ; ------------------------------------------- ;
  ( DEFUN SM (LAMBDA ()
     (SETQ I 22)
     (LOOP
        ( (EQ I 1000) )
        (SETQ A1 (* I I))
        (SETQ A2 (* (+ I 1) (+ I 1)))
        (SETQ A3 (* (+ I 2) (+ I 2)))
        (SETQ A4 (* (+ I 3) (+ I 3)))
        ; ----------------------------------- ;
        (SETQ A5 (* (+ I 4) (+ I 4)))
        (SETQ A6 (* (+ I 5) (+ I 5)))
        (SETQ A7 (* (+ I 6) (+ I 6)))
        ; ------------------------------------- ;
        (SETQ B1 (+ A1 (+ A2 (+ A3 A4))))
        (SETQ B2 (+ A5 (+ A6 A7)))
        ( (EQ B1 B2) (PRINT I) )
        (SETQ I (+ I 1))
     )
  ))
```

Task 8.

```
  ; ------------------------------------------------- ;
  ; Given an integer K>7, find a pair of non-negative
  integers A and B such that K = 3A + 5B ;
  ; ------------------------------------------------- ;
  (DEFUN FOR (LAMBDA (K)
     (SETQ N 100) (PRIN1 "Value of A and B ...: ") (SETQ A 0)
     ( LOOP
         ( (> A N) )
         (SETQ B 0)
         (LOOP
            ( (> B N) )
            ( (EQ K (+ (* A 3) (* B 5)))
                    ((PRIN1 A) (PRIN1 " ") (PRINT B))
            )
            (SETQ B (+ B 1))
         )
         (SETQ A (+ A 1))
     )
  ))
```

Task 9.

```
  ; ------------------------------------------ ;
  ; Define a function FIB(N) that calculates the N-th ;
  ; element of the Fibonacci sequence ;
  ; ------------------------------------------ ;
  ; Imperative programming style ;
  ; ------------------------------------------ ;
  (DEFUN FIB1 (LAMBDA (K))
    (COND
        ( (EQ K 1) 1 )
        ( (EQ K 2) 1 )
        ( T ( (SETQ I 3) (SETQ F1 1)
                (SETQ F2 1) (SETQ F3 2)
                ( LOOP
                    ( (> I K) F3)
                    (SETQ F3 (+ F1 F2))
                    (SETQ F1 F2) (SETQ F2 F3)
                    (SETQ I (+ I 1)))) )
    )
))
; ------------------------------------- ;
; Functional programming style ;
; ------------------------------------- ;
(DEFUN FIB (LAMBDA (K)
    (COND ( (ZEROP K) 0)
          ( (EQ K 1)  1)
          (   T  (+ (FIB (- K 1))
                     (FIB (- K 2))) )
    )
))
```

Task 10.

```
; ------------------------------------------------- ;
; If the entered Fibonacci number K and the number itself are ;
; multiples of five, then return a list of the ;
; Fibonacci number and its number, otherwise NIL ;
; ------------------------------------------------- ;
(DEFUN FIBFIFE (LAMBDA (K)
    (COND ( (AND (ZEROP (CDR (DIVIDE      K  5)))
                 (ZEROP (CDR (DIVIDE (FIB K) 5)))
            )    (LIST (FIB K) K)
          )
          (  T  NIL )
    )
))
; ----------------- ;
(DEFUN FIB (LAMBDA (K)
    (COND
        ( (EQ K 1) 1 )
        ( (EQ K 2) 1 )
        ( T ( (SETQ I 3) (SETQ F1 1)
                 (SETQ F2 1) (SETQ F3 2)
                 ( LOOP
                      ( (> I K) F3 )
                      (SETQ F3 (+ F1 F2))
                      (SETQ F1 F2)
                      (SETQ F2 F3)
                      (SETQ I (+ I 1)))) )
    )
))
```

Task 11.

```
; ---------------------------------------------------- ;
; Convert the number N to the number system with base F ;
; ---------------------------------------------------- ;
(DEFUN YES (LAMBDA (N F))
    (SETQ LST NIL)
    (SETQ REM (CDR (DIVIDE N F)))
    (SETQ I   (CAR (DIVIDE N F)))
    (SETQ LST (CONS REM LST))
    ( LOOP
        (SETQ N I)
        (SETQ REM (CDR (DIVIDE N F)))
        (SETQ I   (CAR (DIVIDE N F)))
        (SETQ LST (CONS REM LST))
        ( (< I F) )
    )
    (PRINT (SETQ LST (CONS I LST)))
))
```

Problem 12.

```
; --------------------------------------------------- ;
; Write the smallest three-digit number that is a multiple of 3, ;
; so that its first digit is 6, and all the digits are ;
; different ;
; --------------------------------------------------- ;
( DEFUN CHCH (LAMBDA ()
    (SETQ I 0)
    (LOOP
       ( (> I 9) )
       (SETQ J 0)
       ( LOOP
           ( (> J 9) )
           (COND
              ( (AND (ZEROP (MOD
                                  (+ (* 10 I)
                                        (+ J 600)) 3))
                     (NOT (EQ I J))
                     (NOT (EQ I 6)) (NOT (EQ J 6)))
                   (PRIN1 6) (PRIN1 I)
                   (PRIN1 J) (PRIN1 " ")) )
           (SETQ J (+ J 1))
       )
       (SETQ I (+ I 1))
    )
))
```

Problem 13.

```
; ----------------------------------------------- ;
; Natural numbers m and n are relatively prime and n < m. ;
; Which number is greater: ;
; (1*m/n) + (2*m/n) +...+ (n*m/n) ;
; or ;
; (1*n/m) + (2*n/m) +...+ (m*n/m)? ;
; ---------------------------------------------- ;
(DEFUN SUMMA (LAMBDA (N M)
    (SETQ S2 0) (SETQ I 1)
    (PRIN1 "Enter M...") (SETQ M (READ))
    (PRIN1 "Enter N...") (SETQ N (READ))
    (LOOP
        ( (> I M) S2 )
        (SETQ S2 (+ S2 (CAR (DIVIDE (* I N) M))))
        (SETQ I (+ I 1))
    )
    (SETQ S1 0) (SETQ I 1)
    (LOOP
       ( (> I N ) S1 )
       (SETQ S1 (+ S1 (CAR (DIVIDE (* I M) N))))
```

```
        (SETQ I (+ I 1))
     )
     (COND
        ( (> S1 S2) (PRINT "First number is greater") )
        ( T (PRINT "The second number is greater") )
     )
     (PRIN1 "Difference...") (- S1 S2)
  ))
```

Task 14.

```
  ; --------------------------------------- ;
  ; Find three-digit numbers equal to the sum ;
  ; of the factorials of their digits ;
  ; --------------------------------------- ;
  ( DEFUN MAIN (LAMBDA ()
     (SETQ N 100)
     (LOOP
        ( (EQ N 999) 999 )
        ( (EQ N (+
                   (FACT (CAR  (STRING-NUM (UNPACK N))))
                   (+
                     (FACT
                        (CADR  (STRING-NUM (UNPACK N))))
                        (FACT (CADDR
                                (STRING-NUM (UNPACK N))))
                      )
                )
          )
          ( (PRINT "Number...") N )
        )
        ( SETQ N (+ N 1) )
     )
  ))
  ; ------------------- ;
  (DEFUN FACT (LAMBDA (X)
     (COND
        ( (ZEROP X) 1 )
        (  T  (* X (FACT (- X 1))) )
     )
  ))
  ; ------------------------- ;
  (DEFUN STRING-NUM(LAMBDA (X) )
  ; Translate a list of digits into a list containing ;
  ; the corresponding single-digit numbers ;
      (COND ( (EQ (LENGTH X) 1)
                  (CONS
                     (POSITION (CAR X) (UNPACK 123456789))
                     NIL) )
            (  T  (CONS
                     (POSITION (CAR X) (UNPACK 123456789))
                     (STRING-NUM (CDR X))) ))
  ))
  ; --------------------------- ;
  (DEFUN POSITION (LAMBDA (X LST)
  ; The POSITION function returns the position of the atom X ;
  ; in a single-level list LST (the first element has ;
  ; number 1). If the element is not in the list, the function ;
  ; returns 0 ;
     (COND ( (NULL LST)       0 )
           ( (EQ X (CAR LST)) 1 )
           ( (MEMBER X LST)
                (+ 1 (POSITION X (CDR LST))) )
           ( T  0 )
      )
  ))
```

Task 15.

```
  ; ---------------------------------------------------- ;
  ; Write a program that, when executed, ;
  ; finds out whether the digit 2 is included in the notation of a given integer ;
  ; number n ;
  ;----------------------------------------------------- ;
  ( DEFUN MAIN (LAMBDA ()
     (SETQ LST (STRING-NUM (UNPACK (READ))))
     (SETQ FLAG 0)
     (LOOP
        ( (NULL LST) FLAG )
        ( (EQ (CAR LST) 2) (SETQ FLAG 1) )
        (  SETQ LST (CDR LST) )
     )
     (COND
        ( (EQ FLAG 1) (PRINT "There is number 2 in the record") )
        ( T (PRINT "No number 2 in record") )
     )
  ))
  ; ------------------------- ;
  (DEFUN STRING-NUM(LAMBDA (X) )
  ; Translate a list of digits into a list containing ;
  ; the corresponding single-digit numbers ;
     (COND ( (NULL X) NIL )
           (  T  (CONS (POSITION (CAR X) (UNPACK 123456789))
                       (STRING-NUM (CDR X))
                 )
           )
     )
  ))
  ; --------------------------- ;
  (DEFUN POSITION (LAMBDA (X LST)
  ; The POSITION function returns the position of the atom X ;
  ; in a single-level list LST (the first element has ;
  ; number 1). If the element is not in the list, the function ;
  ; returns 0 ;
     (COND ( (NULL LST)       0 )
           ( (EQ X (CAR LST)) 1 )
           ( (MEMBER X LST)
                (+ 1 (POSITION X (CDR LST))) )
           ( T  0 )
     )
  ))
```

We have finished covering the basics of the **LISP** programming language.